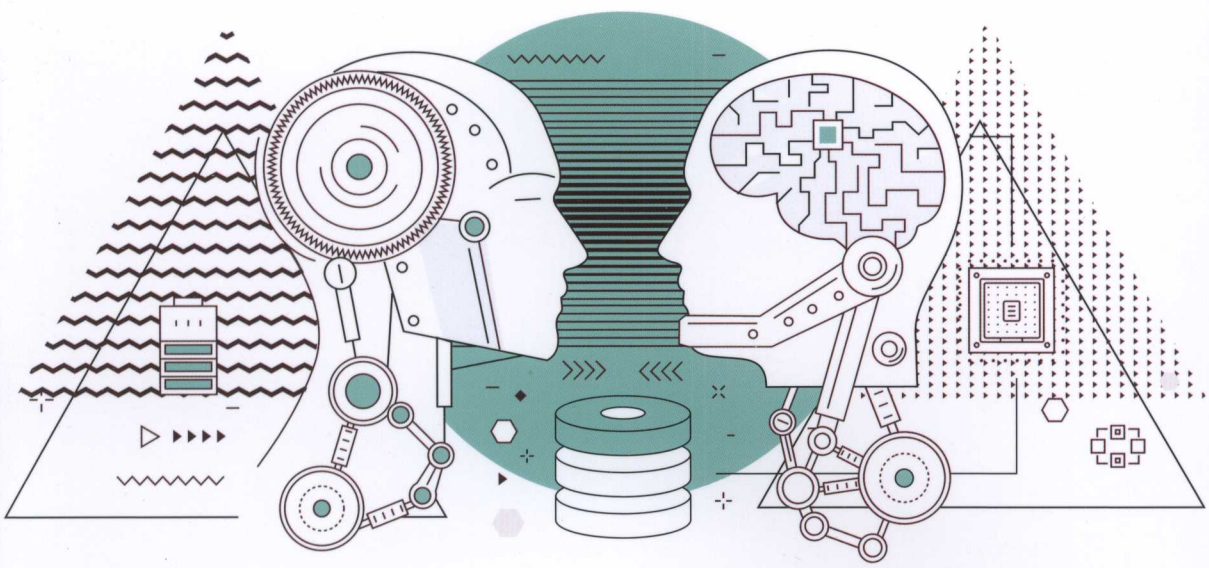
版权注意事项：
1、书籍版权归作者和出版社所有
2、本PDF仅限用于个人获取知识，进行私底下的知识交流
3、PDF获得者不得在互联网上以任何目的进行传播
4、如觉得书籍内容很赞，请购买正版实体书，支持作者
5、请于下载PDF后24小时内删除本PDF。

系统讲解Spark机器学习的技术、原理、算法和组件，以及构建Spark机器学习系统的方法、流程、标准和规范

Inside Explore the Spark Machine Learning

深度实践 Spark机器学习

吴茂贵 郁明敏 朱凤元 张粤磊◎等著



机械工业出版社
China Machine Press

内容简介

本书系统讲解了Spark机器学习的技术、原理、组件、算法，以及构建Spark机器学习系统的方法、流程、标准和规范。此外，还介绍了Spark的深度学习框架TensorFlowOnSpark，以及如何借助它实现卷积神经网络和循环神经网络。

全书共14章，分为4个部分：

第一部分（第1~7章）

主要讲解了Spark机器学习的技术、原理和核心组件，包括Spark ML、Spark ML Pipeline、Spark MLlib，以及如何构建一个Spark机器学习系统。

第二部分（第8~12章）

主要以实例为主，讲解了Spark ML的各种机器学习算法，包括推荐模型、分类模型、聚类模型、回归模型，以及PySpark决策树模型和Spark R朴素贝叶斯模型。

第三部分（第13章）

与之前的批量处理不同，本章以在线数据或流式数据为主，讲解了Spark的流式计算框架Spark Streaming。

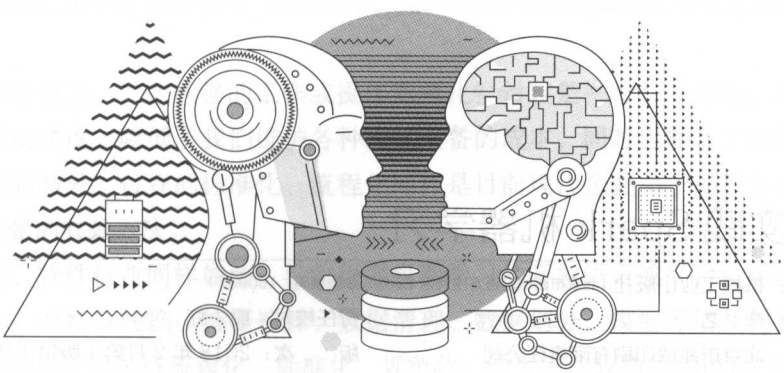
第四部分（第14章）

介绍了Spark深度学习，主要包括TensorFlow的基础知识及它与Spark的整合框架TensorFlowOnSpark。

Inside Explore the Spark Machine Learning

深度实践 Spark机器学习

吴茂贵 郁明敏 朱凤元 张粤磊 杨本法 著



机械工业出版社
China Machine Press

图书在版编目（CIP）数据

深度实践 Spark 机器学习 / 吴茂贵等著 . —北京：机械工业出版社，2018.1

ISBN 978-7-111-58995-2

I. 深… II. 吴… III. 数据处理软件 - 机器学习 IV. TP274

中国版本图书馆 CIP 数据核字（2018）第 015240 号

深度实践 Spark 机器学习

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号 邮政编码：100037）

责任编辑：李 艺　　责任校对：殷 虹

印　　刷：北京市兆成印刷有限责任公司　　版　　次：2018 年 2 月第 1 版第 1 次印刷

开　　本：186mm × 240mm 1/16　　印　　张：15.25

书　　号：ISBN 978-7-111-58995-2　　定　　价：69.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066　　投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259　　读者信箱：hzit@hzbook.com

版权所有 · 侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光 / 邹晓东

Preface 前　言

为什么写这本书

大数据、人工智能正在改变或颠覆各行各业，包括我们的生活。大数据、人工智能方面的人才已经供不应求，但作为人工智能的核心——机器学习，因涉及的知识和技能比较多，除了需要具备一定的数学基础、相关业务知识外，还要求有比较全面的技术储备，如操作系统、数据库、开发语言、数据分析工具、大数据计算平台等，无形中提高了机器学习的门槛。如何降低机器学习的门槛，让更多有志于机器学习、人工智能的人能更方便或顺畅地使用、驾驭机器学习？

很多企业也正在考虑和处理这方面的问题，本书也希望借 Spark 技术在这方面做一些介绍或总结。

如何使原本复杂、专业性强的工作或操作简单化？封装是一个有效方法。封装降低了我们操作照相机的难度、降低了我们维护各种现代设备的成本，同时也提升了我们使用这些设备的效率。除封装外，过程的标准化、流程化同样是目前现代企业用于提升生产效率，降低成本，提高质量的有效方法。

硬件如此，软件行业同样如此。目前很多机器学习的开发语言或平台，正在这些方面加大力度，比如：对特征转换、特征选择、数据清理、数据划分、模型评估及优化等算法的封装；对机器学习过程的进行流程化、标准化、规范化；给大家比较熟悉的语言或工具提供 API 等方法或措施，以简化机器学习中间过程，缩短整个开发周期，使我们能更从容地应对市场的变化。Spark 在这方面可谓后来居上，尤其是最近发布的版本，明显加大了这方面的力度，我们可以从以下几个方面看出这种趋势：

1）Spark 机器学习的 API，正在由基于 RDD 过渡到基于 Dataset 或 DataFrame，基于 RDD 的 API 在 Spark2.2 后处于维护阶段，Spark3.0 后将停止使用（来自 Spark 官网）；

2）建议大家使用 Spark ML，尤其是它的 Pipeline；

3）增加大量特征选择、特征转换、模型选择和优化等算法；

4）丰富、增强 Spark 与 Java、Python、R 的 API，使其更通用。

SKLearn、Spark 等机器学习平台或工具在这方面都处于领先的地位，我们也希望借助本书，把 Spark 在这方面的有关内容介绍给大家，使大家可以少走些弯路。

此外，Spark 目前主要涉及常用机器学习算法，缺乏对一般神经网络的支持，更不用说深度学习了，这好像也是目前 Spark 的一个不足。不过好消息是：雅虎把深度学习框架 TensorFlow 与 Spark 整合在一起，而且开源了这些代码。为弥补广大 Spark 爱好者的上述缺憾，本书介绍了 TensorFlowOnSpark，其中包括深度学习框架 TensorFlow 的基础知识及使用卷积神经网络、循环神经网络等的一些实际案例。

另外，我们提供了与本书环境完全一致的免费云操作环境，这样一来是希望节约您的宝贵时间，二来是希望能通过真正的实战，给您不一样的体验和收获！总之，我们希望能使更多有志于大数据、人工智能的朋友加入这个充满生机、前景广阔的行业中来。

本书特色

本书最大特点就是注重实战！或许有读者会问，能从哪几个方面体现出来？

1）介绍了目前关于机器学习的新趋势，并分析了如何使用 Pipeline 使机器学习过程流程化。

2）简介了机器学习的一般框架 Spark、深度学习框架 Tensorflow 及把两者整合在一起的框架 TensorflowOnSpark。

3）提供可操作、便执行及具有实战性的项目及其详细代码。

4）提供与书完全一致的云操作环境，而且这个环境可以随时随地使用实操环境，登录地址为 http://www.feiguyun.com/spark/support。

5）除了代码外，还附有一些必要的架构或原理说明，便于大家能从一个更高的角度来理解把握相关问题。

总之，希望你通过阅读本书，不但可以了解很多内容或代码，更可以亲自运行或调试这些代码，从而带来新的体验和收获！

读者对象

- ❑ 对大数据、机器学习感兴趣的广大在校、在职人员。
- ❑ 对 Spark 机器学习有一定基础，欲进一步提高开发效率的人员。
- ❑ 熟悉 Python、R 等工具，希望进一步拓展到 Spark 机器学习的人。

❑ 对深度学习框架 TensorFlow 及其拓展感兴趣的读者。

如何阅读本书

本书正文共 14 章，从内容结构来看，可以分为四部分。

第一部分为第 1 ～ 7 章，主要介绍了机器学习的一些基本概念，包括如何构建一个 Spark 机器学习系统，Spark ML 主要特点，Spark ML 中流水线（Pipeline），ML 中大量特征选取、特征转换、特征选择等函数或方法，同时简单介绍了 Spark MLlib 的一些基础知识。

第二部分为第 8 ～ 12 章，主要以实例为主，具体说明如何使用 Spark ML 中 Pipeline 的 Stage，以及如何把这 Stage 组合到流水线上，最后通过评估指标，优化模型。

第三部分即第 13 章，与之前的批量处理不同，这一章主要以在线数据或流式数据为主，介绍 Spark 的流式计算框架 Spark Streaming。

第四部分即第 14 章，为深度学习框架，主要包括 TensorFlow 的基础知识及它与 Spark 的整合框架 TensorFlowOnSpark。

此外，书中的附录部分还提供了线性代数、概率统计及 Scala 的基础知识，以帮助读者更好地掌握机器学习的相关内容。

勘误和支持

除封面署名外，参加本书编写、环境搭建的人还有杨本法、张魁、刘未昕等、杨本法负责第 12 章 Spark R 的编写，张魁、刘未昕负责后台环境的搭建和维护。由于笔者水平有限，加之编写时间仓促，书中难免出现错误或不准确的地方。恳请读者批评指正，你可以通过访问 http://www.feiguyun.com 留下宝贵意见。也可以通过微信（wumg3000）或 QQ（1715408972）给我们反馈。非常感谢你的支持和帮助。

致谢

在本书编写过程中，得到很多在校老师和同学的支持！感谢上海大学机电工程与自动化学院的王佳寅老师及黄文成、杨中源、熊奇等同学，上海理工管理学院的张帆老师，上海师大数理学院的田红炯、李昭祥老师，华师大的王旭同学，博世王冬，飞谷云小伙伴等提供的支持和帮助。

感谢机械工业出版社的杨福川、李艺老师给予本书的大力支持和帮助。

感谢参与本书编写的其他作者及提供支持的家人们，谢谢你们！

目　录 *Contents*

前言

第1章　了解机器学习 ····· 1

1.1　机器学习的定义 ····· 1

1.2　大数据与机器学习 ····· 2

1.3　机器学习、人工智能及深度学习 ··· 2

1.4　机器学习的基本任务 ····· 3

1.5　如何选择合适算法 ····· 4

1.6　Spark 在机器学习方面的优势 ····· 5

1.7　小结 ····· 5

第2章　构建Spark机器学习系统 ····· 6

2.1　机器学习系统架构 ····· 6

2.2　启动集群 ····· 7

2.3　加载数据 ····· 9

2.4　探索数据 ····· 10

2.4.1　数据统计信息 ····· 10

2.4.2　数据质量分析 ····· 11

2.4.3　数据特征分析 ····· 12

2.4.4　数据的可视化 ····· 17

2.5　数据预处理 ····· 19

2.5.1　数据清理 ····· 20

2.5.2　数据变换 ····· 21

2.5.3　数据集成 ····· 22

2.5.4　数据归约 ····· 23

2.6　构建模型 ····· 25

2.7　模型评估 ····· 26

2.8　组装 ····· 30

2.9　模型选择或调优 ····· 30

2.9.1　交叉验证 ····· 31

2.9.2　训练 – 验证切分 ····· 32

2.10　保存模型 ····· 32

2.11　小结 ····· 33

第3章　ML Pipeline原理与实战 ····· 34

3.1　Pipeline 简介 ····· 34

3.2　DataFrame ····· 35

3.3　Pipeline 组件 ····· 36

3.4　Pipeline 原理 ····· 37

3.5　Pipeline 实例 ····· 38

3.5.1　使用 Estimator、Transformer 和 Param 的实例 ····· 38

3.5.2　ML 使用 Pipeline 的实例 ····· 40

3.6　小结 ····· 41

第4章 特征提取、转换和选择 ····· 42
4.1 特征提取 ····· 42
4.1.1 词频—逆向文件频率（TF-IDF） ····· 42
4.1.2 Word2Vec ····· 43
4.1.3 计数向量器 ····· 44
4.2 特征转换 ····· 45
4.2.1 分词器 ····· 45
4.2.2 移除停用词 ····· 46
4.2.3 n-gram ····· 47
4.2.4 二值化 ····· 48
4.2.5 主成分分析 ····· 48
4.2.6 多项式展开 ····· 50
4.2.7 离散余弦变换 ····· 50
4.2.8 字符串—索引变换 ····· 51
4.2.9 索引—字符串变换 ····· 53
4.2.10 独热编码 ····· 54
4.2.11 向量—索引变换 ····· 57
4.2.12 交互式 ····· 58
4.2.13 正则化 ····· 59
4.2.14 规范化 ····· 60
4.2.15 最大值—最小值缩放 ····· 60
4.2.16 最大值—绝对值缩放 ····· 61
4.2.17 离散化重组 ····· 62
4.2.18 元素乘积 ····· 63
4.2.19 SQL 转换器 ····· 64
4.2.20 向量汇编 ····· 65
4.2.21 分位数离散化 ····· 66
4.3 特征选择 ····· 67
4.3.1 向量机 ····· 67
4.3.2 R 公式 ····· 69
4.3.3 卡方特征选择 ····· 70
4.4 小结 ····· 71

第5章 模型选择和优化 ····· 72
5.1 模型选择 ····· 72
5.2 交叉验证 ····· 73
5.3 训练验证拆分法 ····· 75
5.4 自定义模型选择 ····· 76
5.5 小结 ····· 78

第6章 Spark MLlib基础 ····· 79
6.1 Spark MLlib 简介 ····· 80
6.2 Spark MLlib 架构 ····· 81
6.3 数据类型 ····· 82
6.4 基础统计 ····· 84
6.4.1 摘要统计 ····· 84
6.4.2 相关性 ····· 84
6.4.3 假设检验 ····· 85
6.4.4 随机数据生成 ····· 85
6.5 RDD、Dataframe 和 Dataset ····· 86
6.5.1 RDD ····· 86
6.5.2 Dataset/DataFrame ····· 87
6.5.3 相互转换 ····· 88
6.6 小结 ····· 89

第7章 构建Spark ML推荐模型 ····· 90
7.1 推荐模型简介 ····· 91
7.2 数据加载 ····· 92
7.3 数据探索 ····· 94
7.4 训练模型 ····· 94
7.5 组装 ····· 95

7.6 评估模型 …… 96
7.7 模型优化 …… 96
7.8 小结 …… 98

第8章 构建Spark ML分类模型 …… 99

8.1 分类模型简介 …… 99
8.1.1 线性模型 …… 100
8.1.2 决策树模型 …… 101
8.1.3 朴素贝叶斯模型 …… 102
8.2 数据加载 …… 102
8.3 数据探索 …… 103
8.4 数据预处理 …… 104
8.5 组装 …… 109
8.6 模型优化 …… 110
8.7 小结 …… 113

第9章 构建Spark ML回归模型 …… 114

9.1 回归模型简介 …… 115
9.2 数据加载 …… 115
9.3 探索特征分布 …… 117
9.4 数据预处理 …… 120
9.4.1 特征选择 …… 121
9.4.2 特征转换 …… 121
9.5 组装 …… 122
9.6 模型优化 …… 124
9.7 小结 …… 126

第10章 构建Spark ML聚类模型 …… 127

10.1 K-means 模型简介 …… 128
10.2 数据加载 …… 129
10.3 探索特征的相关性 …… 129
10.4 数据预处理 …… 131
10.5 组装 …… 132
10.6 模型优化 …… 134
10.7 小结 …… 136

第11章 PySpark 决策树模型 …… 137

11.1 PySpark 简介 …… 138
11.2 决策树简介 …… 139
11.3 数据加载 …… 140
11.3.1 原数据集初探 …… 140
11.3.2 PySpark 的启动 …… 142
11.3.3 基本函数 …… 142
11.4 数据探索 …… 143
11.5 数据预处理 …… 143
11.6 创建决策树模型 …… 145
11.7 训练模型进行预测 …… 146
11.8 模型优化 …… 149
11.8.1 特征值的优化 …… 149
11.8.2 交叉验证和网格参数 …… 152
11.9 脚本方式运行 …… 154
11.9.1 在脚本中添加配置信息 …… 154
11.9.2 运行脚本程序 …… 154
11.10 小结 …… 154

第12章 SparkR朴素贝叶斯模型 …… 155

12.1 SparkR 简介 …… 156
12.2 获取数据 …… 157
12.2.1 SparkDataFrame 数据结构说明 …… 157
12.2.2 创建 SparkDataFrame …… 157
12.2.3 SparkDataFrame 的常用操作 … 160
12.3 朴素贝叶斯分类器 …… 162
12.3.1 数据探查 …… 162

12.3.2 对原始数据集进行转换……163
12.3.3 查看不同船舱的生还率差异……163
12.3.4 转换成 SparkDataFrame 格式的数据……165
12.3.5 模型概要……165
12.3.6 预测……165
12.3.7 评估模型……166
12.4 小结……167

第13章 使用Spark Streaming 构建在线学习模型……168

13.1 Spark Streaming 简介……168
13.1.1 Spark Streaming 常用术语…169
13.1.2 Spark Streaming 处理流程…169
13.2 Dstream 操作……170
13.2.1 Dstream 输入……170
13.2.2 Dstream 转换……170
13.2.3 Dstream 修改……171
13.2.4 Dstream 输出……172
13.3 Spark Streaming 应用实例……172
13.4 Spark Streaming 在线学习实例…174
13.5 小结……175

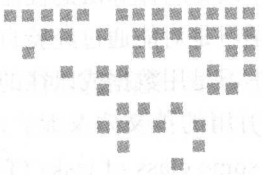
第14章 TensorFlowOnSpark详解…176

14.1 TensorFlow 简介……176
14.1.1 TensorFlow 的安装……177
14.1.2 TensorFlow 的发展……177
14.1.3 TensorFlow 的特点……177
14.1.4 TensorFlow 编程模型……178
14.1.5 TensorFlow 常用函数……180
14.1.6 TensorFlow 运行原理……181
14.1.7 TensorFlow 系统架构……182
14.2 TensorFlow 实现卷积神经网络…183
14.2.1 卷积神经网络简介……183
14.2.2 卷积神经网络的发展历程…184
14.2.3 卷积神经网络的网络结构…186
14.2.4 TensorFlow 实现卷积神经网络……186
14.3 TensorFlow 实现循环神经网络…191
14.3.1 循环神经网络简介……191
14.3.2 LSTM 循环神经网络简介…192
14.3.3 LSTM 循环神经网络分步说明……193
14.3.4 TensorFlow 实现循环神经网络……194
14.4 分布式 TensorFlow……198
14.4.1 客户端、主节点和工作节点间的关系……198
14.4.2 分布式模式……198
14.4.3 在 Pyspark 集群环境运行 TensorFlow……199
14.5 TensorFlowOnSpark 架构……202
14.6 TensorFlowOnSpark 安装……203
14.7 TensorFlowOnSpark 实例……204
14.7.1 TensorFlowOnSpark 单机模式实例……204
14.7.2 TensorFlowOnSpark 集群模式实例……207
14.8 小结……208

附录A 线性代数……209

附录B 概率统计……214

附录C Scala基础……220



第 1 章 Chapter 1

了解机器学习

大数据、人工智能是目前大家谈论比较多的话题，它们的应用也越来越广泛，与我们的生活关系也越来越密切，影响也越来越深远，其中很多已进入寻常百姓家，如无人机、网约车、自动导航、智能家电、电商推荐、人机对话机器人等。

大数据是人工智能的基础，而使大数据转变为知识或生产力，离不开机器学习（Machine Learning），可以说机器学习是人工智能的核心，是使机器具有类似人的智能的根本途径。

本章主要介绍与机器学习有关的概念，机器学习与大数据、人工智能间的关系，机器学习常用架构及算法等，具体如下：

- 机器学习的定义
- 大数据与机器学习
- 机器学习与人工智能、深度学习
- 机器学习的基本任务
- 如何选择合适算法
- Spark 在机器学习方面的优势

1.1 机器学习的定义

机器学习是什么？是否有统一或标准定义？目前好像没有，即使在机器学习的专业领域，也没有一个被广泛认可的定义。在维基百科上对机器学习有以下几种定义：

（1）机器学习是一门人工智能的科学，该领域的主要研究对象是人工智能，特别是如

何在经验学习中改善具体算法的性能。

（2）机器学习是对能通过经验自动改进的计算机算法的研究。

（3）机器学习是用数据或以往的经验来优化计算机程序的性能标准。

一种经常引用的英文定义是：A computer program is said to learn from experience (E) with respect to some class of tasks (T) and performance(P) measure, if its performance at tasks in T, as measured by P, improves with experience E。

可以看出机器学习强调三个关键词：算法、经验、性能，其处理过程如图 1-1 所示。

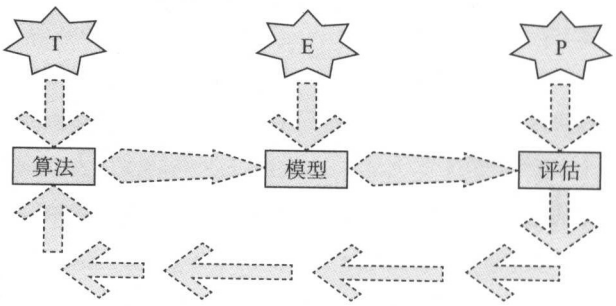


图 1-1 机器学习处理流程

图 1-1 表明机器学习是使数据通过算法构建出模型，然后对模型性能进行评估，评估后的指标如果达到要求就用这个模型测试新数据，如果达不到要求就要调整算法重新建立模型，再次进行评估，如此循环往复，最终获得满意结果。

1.2 大数据与机器学习

我们已进入大数据时代，产生数据的能力迅速增长，如互联网、移动互联网、物联网、成千上万的传感器、穿戴设备、GPS 等都会产生大量数据，存储数据、处理数据等能力也得到了几何级数的提升，如利用 Hadoop、Spark 技术为我们存储、处理大数据提供有效方法。

数据就是信息，就是依据，其背后隐含了大量不易被我们感官识别的信息、知识、规律等，如何揭示这些信息、规则、趋势，正成为当下能给企业带来高回报的热点。

而机器学习的任务，就是要在大数据量的基础上，发掘其中蕴含的有用信息。其处理的数据越多，机器学习就越能体现出优势，以前很多用机器学习解决不了或处理不好的问题，通过大数据可以得到很好解决，性能也会大幅提升，如语言识别、图像识别、天气预测等。

1.3 机器学习、人工智能及深度学习

“人工智能”和“机器学习”这两个科技术语如今广为流传，已成为当下的热词，然而，

它们有何区别？又有哪些相同或相似的地方？虽然人工智能和机器学习高度相关，但却并不尽相同。

人工智能是计算机科学的一个分支，目的是开发一种拥有智能行为的机器，目前很多大公司都在努力开发这种机器学习技术，努力让计算机学会人类的行为模式，以便推动很多人眼中的下一场技术革命——让机器像人类一样“思考”。

过去 10 年，机器学习为我们带来了无人驾驶汽车、实用的语音识别、有效的网络搜索等。接下来它将如何改变我们的生活？在哪些领域最先发力？让我们拭目以待。

有一点需要注意，对很多机器学习来说，特征提取不是一件简单的事情。在一些复杂问题上，要想通过人工的方式设计有效的特征集合，往往要花费很多的时间和精力。

作为机器学习的一个分支，深度学习解决的核心问题之一就是自动将简单的特征组合成更加复杂的特征，并利用这些组合特征解决问题。它除了可以学习特征和任务之间的关联以外，还能自动从简单特征中提取更加复杂的特征。图 1-2 展示了深度学习和传统机器学习在流程上的差异。深度学习算法可以从数据中学习更加复杂的特征表达，使得最后一步权重学习变得更加简单且有效。

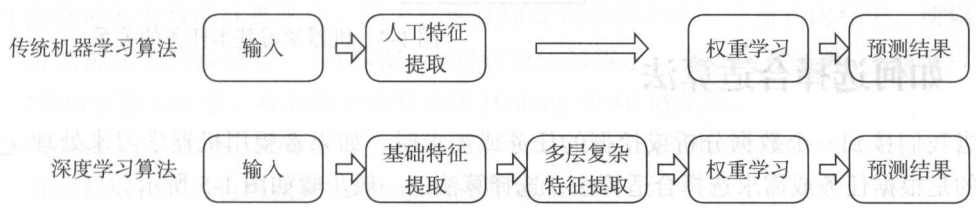


图 1-2　机器学习与深度学习流程对比

前面我们分别介绍了机器学习、人工智能及深度学习，那么它们的关系如何？

人工智能、机器学习和深度学习是紧密相关的几个领域。图 1-3 说明了它们之间的大致关系。人工智能是一类非常广泛的问题，机器学习是解决这类问题的一个重要手段，深度学习则是机器学习的一个分支。在很多人工智能问题上，深度学习的方法突破了传统机器学习方法的瓶颈，推动了人工智能领域的快速发展。

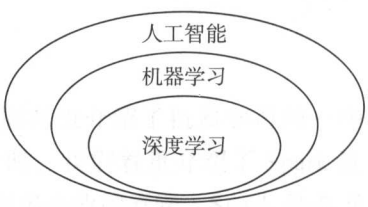


图 1-3　人工智能、机器学习与深度学习间的关系

1.4　机器学习的基本任务

机器学习基于数据，并以此获取新知识、新技能。它的任务有很多，分类是其基本任务之一。所谓分类，就是将新数据划分到合适的类别中，一般用于类别型的目标特征。如果目标特征为连续型，则往往采用回归方法。回归是对新目标特征进行预测，是机器学习中使用非常广泛的方法之一。

分类和回归，都是先根据标签值或目标值建立模型或规则，然后利用这些带有目标值的数据形成的模型或规则，对新数据进行识别或预测。这两种方法都属于监督学习。与监督学习相对的是无监督学习，无监督学习不指定目标值或预先无法知道目标值，它可以把相似或相近的数据划分到相同的组里，聚类就是解决这一类问题的方法之一。

除了监督学习、无监督学习这两种最常见的任务外，还有半监督学习、强化学习等，这里我们就不展开了，图 1-4 展示了这些基本任务间的关系。

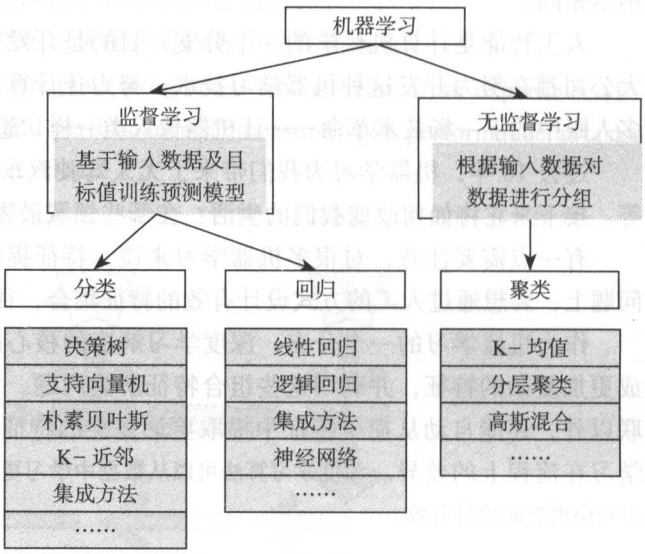


图 1-4 机器学习基本任务的关系

1.5 如何选择合适算法

当我们接到一个数据分析或挖掘的任务或需求时，如果希望用机器学习来处理，首先要做的是根据任务或需求选择合适算法，选择算法的一般步骤如图 1-5 所示。

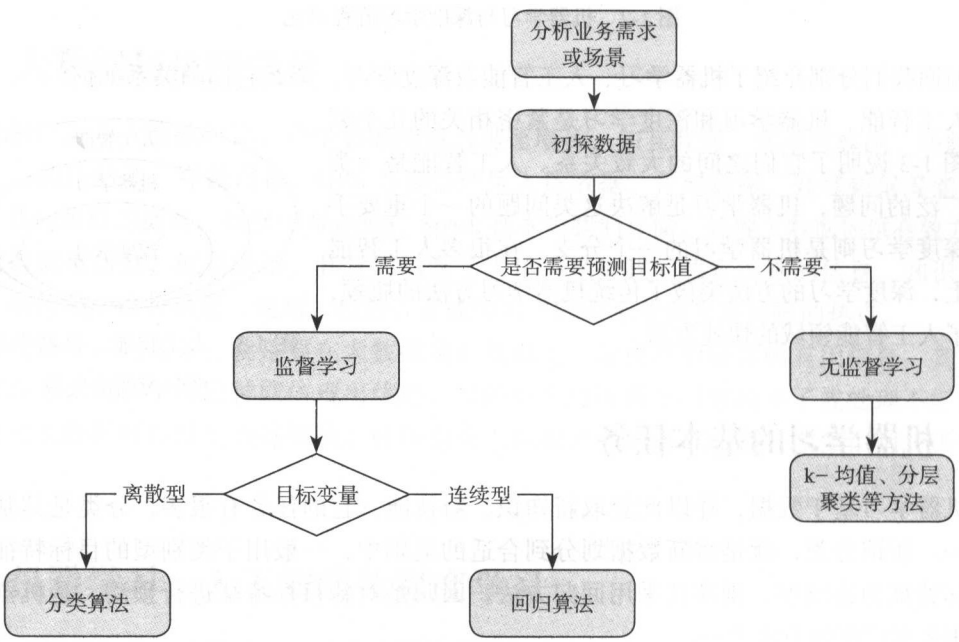


图 1-5 选择算法的一般步骤

充分了解数据及其特性，有助于我们更有效地选择机器学习算法。采用以上步骤在一定程度上可以缩小算法的选择范围，使我们少走些弯路，但在具体选择哪种算法方面，一般并不存在最好的算法或者可以给出最好结果的算法。在实际做项目的过程中，这个过程往往需要多次尝试，有时还要尝试不同算法。不过先用一种简单熟悉的方法，然后，在这个基础上不断优化，时常能收获意想不到的效果。

1.6　Spark 在机器学习方面的优势

在大数据基础上进行机器学习，需要处理全量数据并进行大量的迭代计算，这要求机器学习平台具备强大的处理能力。Spark 与 Hadoop 兼容，它立足于内存计算，天然适用于迭代式计算。Spark 是一个大数据计算平台，其具体有以下优势：

- 完整的大数据生态系统：大家熟悉的 SQL 式操作组件 Spark SQL，功能强大、性能优良的机器学习库 Spark MLlib，用于图像处理的 SparkGraphx 及用于流式处理的 SparkStreaming 等。
- 高性能的大数据计算平台：因为数据被加载到集群主机的分布式内存中，所以数据可以被快速转换迭代，并缓存后续的频繁访问需求。基于内存运算，Spark 可以比 Hadoop 快 100 倍，在磁盘中运算也比 Hadoop 快 10 倍左右。
- 与 Hadoop、Hive、HBase 等无缝连接：Spark 可以直接访问 Hadoop、Hive、HBase 等的数据，同时也可使用 Hadoop 的资源管理器。
- 易用、通用、好用：Spark 编程非常高效、简洁，支持多种语言的 API，如 Scala、Java、Python、R、SQL 等，同时提供类似于 Shell 的交互式开发环境 REPL。

1.7　小结

本章简单介绍了机器学习与大数据、人工智能的关系，同时也介绍了机器学习的一些基本任务和如何选择合适算法等问题。在选择机器学习平台时，我们着重介绍了 Spark 这样一个大数据平台的集大成者，它有很多优势，而且得到了很多企业的青睐。Spark 是本书的主要介绍对象，下一章我们将介绍如何构建一个 Spark 机器学习系统。



Chapter 2 第 2 章

构建 Spark 机器学习系统

构建机器学习系统的方法，根据业务需求和使用工具的不同，可能会有些区别，不过主要流程差别不大，基本包括数据抽取、数据探索、数据处理、建立模型、训练模型、评估模型、优化模型、部署模型等阶段。在构建系统前，我们需要考虑系统的扩展性，与其他系统的整合，系统升级及处理方式等。本章我们主要介绍基于 Spark 机器学习的架构设计或系统构建的一般步骤，以及需要注意的一些问题。

构建 Spark 机器学习系统的一般步骤如下：

- ❑ 介绍系统架构
- ❑ 启动集群
- ❑ 加载数据
- ❑ 探索数据
- ❑ 数据预处理
- ❑ 构建模型
- ❑ 模型评估
- ❑ 模型优化
- ❑ 模型保存

2.1 机器学习系统架构

Spark 发展非常快，到我们着手编写本书时，Spark 已升级为 2.1 版。自 2.0 以后，Spark 大大增强了数据流水线的内容。数据流水线的思路与 SKLearn 非常相似，我想这种思

路或许是未来的一个趋势，使机器学习的流程标准化、规范化、流程化，将很多原来需要自己编写的代码封装成可直接调用的模块或函数，模型评估、调优这些任务也可实现了更高的封装，大大降低机器学习的门槛。

Spark 机器学习系统的架构图如图 2-1 所示，其中数据探索与预处理、训练及测试算法或建模阶段可以组装成流水线方式，模型评估及优化阶段可以采用自动化方式。

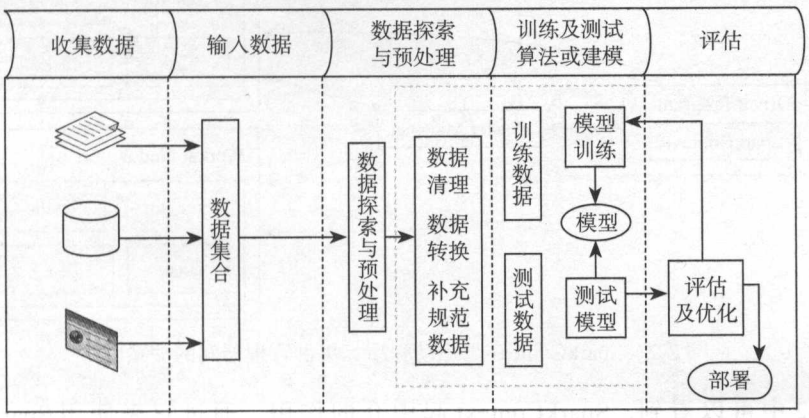


图 2-1　Spark 机器学习系统的架构图

2.2　启动集群

对于 Spark 集群的安装配置，这里不做详细介绍，对 Spark 集群的安装配置感兴趣的读者，可参考由我们编写的《自己动手做大数据系统》。

常见的 Spark 运行方式有本地模式、集群模式。本地模式所有的处理都运行在同一个 JVM 中，而后者，可以运行在不同节点上。具体运行模式如表 2-1 所示。

表 2-1　Spark 运行模式

运行模式	含义
local	使用单线程在本机上运行 Spark 任务
local[K]	使用 *K* 个工作线程在本机上运行 Spark。*K* 值最好小于等于 CPU 的核数
local[*]	使用和 CPU 核数相同的线程数在本机上运行 Spark
spark://host:port	Standalone 模式运行，host 是集群中 Master 节点机器名，port 是端口号，默认是 7077
mesos://host:port	连接到 Mesos 集群运行任务，port 端口号默认是 5055
yarn-client	以客户端方式连接到 YARN 集群
yarn-cluster	以集群方式连接到 YARN 集群

本书主要以 Spark Standalone（独立模式）为例，如果想以其他模式运行，只要改动对应参数即可。

Spark 支持 Scala 或 Python 的 REPL（Read-Eval-Print-Loop，交互式 shell）来进行交互

式程序编写，交互式编程在输入的代码执行后立即能看到结果，非常友好和方便。

在 2.0 之前的 Spark 版本中，Spark shell 会自动创建一个 SparkContext 对象 sc。SparkContext 与驱动程序（Driver Program）和集群管理器（Cluster Manager）间的关系如图 2-2 所示。

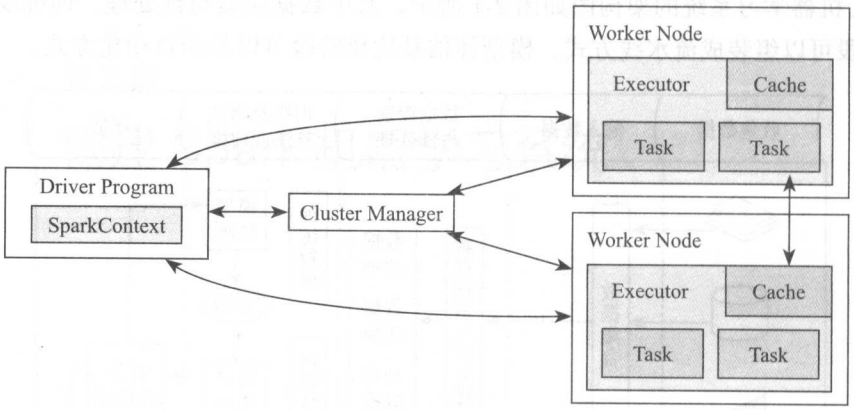


图 2-2 SparkContext 与驱动程序、集群管理器间的关系图

从图 2-2 中可以看到，SparkContext 起中介的作用，通过它来使用 Spark 其他的功能。每一个 JVM 都有一个对应的 SparkContext，Driver Program 通过 SparkContext 连接到集群管理器来实现对集群中任务的控制。Spark 配置参数的设置以及对 SQLContext、HiveContext 和 StreamingContext 的控制也要通过 SparkContext。

不过在 Spark 2.0 中引入了 SparkSession 对象（spark），运行 Spark shell 则会自动创建一个 SparkSession 对象，在输入 spark 时就会发现它已经存在了（见图 2-3）、SparkConf、SparkContext 和 SQLContext 都已经被封装在 SparkSession 当中，它为用户提供了一个统一的切入点，同时也提供了各种 DataFrame 和 Dataset 的 API，大大降低了学习 Spark 的难度。

```
[hadoop@master ~]$ spark-shell --master spark://master:7077
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/03/14 08:26:30 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using
pplicable
17/03/14 08:26:31 WARN Utils: Service 'SparkUI' could not bind on port 4040. Attempting port 4041.
17/03/14 08:26:31 WARN Utils: Service 'SparkUI' could not bind on port 4041. Attempting port 4042.
17/03/14 08:26:34 WARN HiveConf: HiveConf of name hive.metastore.local does not exist
Spark context Web UI available at http://192.168.1.112:4042
Spark context available as 'sc' (master = spark://master:7077, app id = app-20170314082632-0004).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.1.0
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60)
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

图 2-3 启动 Spark shell 界面

图 2-3 所示是启动 Spark 集群的界面，编程语言是 Scala，如果希望使用 Python 为编辑语句，该如何启动呢？运行 Pyspark 即可，如图 2-4 所示。

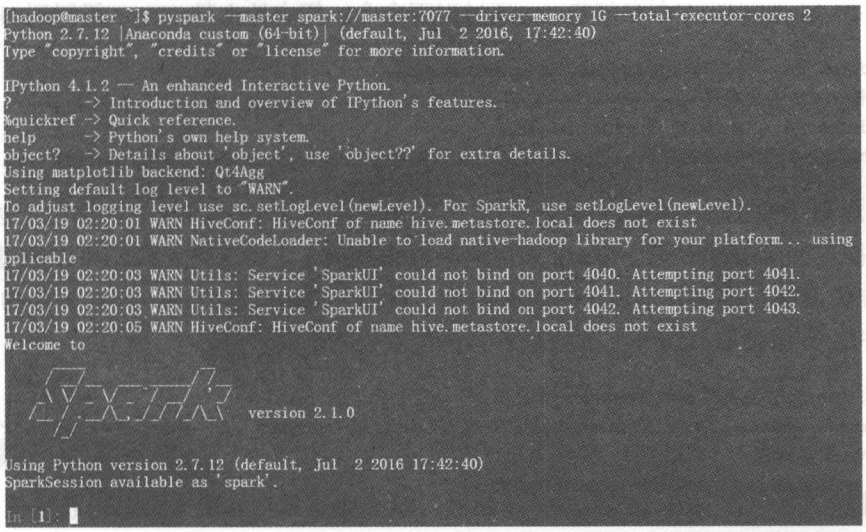


图 2-4　启动 PySpark 的客户端

2.3　加载数据

这里以 MovieLens 100k（http://files.grouplens.org/datasets/movielens/ml-100k.zip）数据集中的用户数据（u.data）为例，首先在本地查看数据的基本信息，然后把本地文件复制到 HDFS 上，Spark 或 PySpark 会读取 HDFS 上的数据。

查看 u.user 文件的基本信息、数据样例、总记录数等信息。

```
$ head -3 u.user
1|24|M|technician|85711
2|53|F|other|94043
3|23|M|writer|32067
$ cat u.user |wc -l
943
```

u.user 用户数据每列的含义为 user id | age | gender | occupation | zip code，即用户 ID、用户年龄、用户性别、用户职位、所在地邮编等信息，列间的分隔符为竖线（|），共有 943 条记录。

如何把用户信息复制到 HDFS 上？首先，查看当前 HDFS 的目录信息。

```
$ hadoop fs -ls /u01/bigdata/
Found 2 items
drwxr-xr-x   - hadoop supergroup          0 2017-02-07 03:20 /u01/bigdata/data
```

```
drwxr-xr-x   - hadoop supergroup          0 2016-07-20 09:16 /u01/bigdata/hive
```

由此可知在 HDFS 上已有 /u01/bigdata/data 目录（如果没有目录可以通过 hadoop fs -mkdire /u01/bigdata/data 命令创建。），通过以下命令，把本地文件 u.user 复制到 HDFS 上。

```
$ hadoop fs -put u.user /u01/bigdata/data
// 查看 HDFS 上的文件
$ hadoop fs -ls /u01/bigdata/data
-rw-r--r--  1 hadoop supergroup    22628 2017-03-18 13:37 /u01/bigdata/data/u.user
```

把电影评级数据（u.data）、电影数据（u.item）等复制到 HDFS 的方法与上边的相同，把本地数据复制到 HDFS 后，Spark 如何读取加载 HDFS 上的文件？我们可以通过 Spark 的 textFile 方法读取。这里我们以 PySpark 为例，启动 PySpark 客户端，导入需要的包，然后通过 textFile 方法读取 HDFS 上的数据，具体请看以下示例：

```
### 以 Spark 独立模式启动 Pyspark 客户端
pyspark --master spark://master:7077 --driver-memory 1G --total-executor-cores 2
### 导入需要的包
from pyspark.sql import SparkSession
from pyspark.sql import Row
## 初始化 sparkSession
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
### 加载数据，并处理分割符数据
sc = spark.sparkContext
userrdd = sc.textFile("hdfs://master:9000/u01/bigdata/data/u.user").map(lambda
line: line.split("|"))
### 利用反射机制推断模式 (Schema)，把 dataframe 注册为一个 table
df = userrdd.map(lambda fields: Row(userid=fields[0], age=int(fields[1]),gender=
     fields[2],occupation=fields[3],zip=fields[4]))
schemauser = spark.createDataFrame(df)
schemauser.createOrReplaceTempView("user")
```

2.4 探索数据

生产环境中数据往往包含很多脏数据，如缺失数据、不一致、不规范、奇异数据等，所以在数据加载后且在数据建模前，需要对数据进行分析或探索，尤其面对大数据，了解数据的统计信息、数据质量、数据特征等，为数据处理、数据建模提供重要依据。在进行这些数据分析时，如果能实现数据的可视化，当然更利于我们理解数据。

2.4.1 数据统计信息

加载数据后，首先要关注数据的统计信息，有了数据统计信息，我们对数据就有了一

个大致了解，如数据特征的最大值、最小值、平均值、分位数、方差等。这些信息有助于我们理解数据质量、数据构成，为数据预处理提供重要依据。查看用户各字段的统计信息：

```
schemauser.describe("userid","age", "gender","occupation","zip").show()
+-------+-----------------+-----------------+------+-------------+-----------------+
|summary|           userid|              age|gender|   occupation|              zip|
+-------+-----------------+-----------------+------+-------------+-----------------+
|  count|              943|              943|   943|          943|              943|
|   mean|            472.0|34.05196182396607|  null|         null| 50868.78810810811|
| stddev|272.3649512449549|12.19273973305903|  null|         null|30891.373254138158|
|    min|                1|                7|     F|administrator|            00000|
|    max|               99|               73|     M|       writer|            Y1A6B|
+-------+-----------------+-----------------+------+-------------+-----------------+
```

从以上统计可以看出，用户表总记录数为 943 条，年龄最小为 7 岁，最大为 73 岁，平均年龄为 34 岁。

2.4.2　数据质量分析

数据质量分析是数据探索阶段重要一环，数据不是完美的，往往存在缺少数据的情况，还可能包含不一致数据、异常数据、噪声数据等。没有可信的数据，再好的模型性能都不太可能好，正所谓“垃圾进，垃圾出”。

数据质量方面的分析，主要包括以下几个方面：

- ❑ 缺失值；
- ❑ 异常值；
- ❑ 不一致的值；
- ❑ 错误数据。

本节以一份某酒店的销售额数据为例，来说明在数据探索中，对数据质量的一般分析方法。

```
## 以 Spark 独立模式，启动 Pyspark 客户端
pyspark --master spark://master:7077 --driver-memory 1G --total-executor-cores 2
### 导入需要的库
import pandas as pd
import matplotlib.pyplot as plt
### 加载数据，使用标题行
df=pd.read_csv("/home/hadoop/data/catering_sale.csv",header=0)
## 查看 df 的统计信息
df.count()    ## 统计非空值记录数
sale_date    200
sale_amt     198   ### 说明 sale_amt 有两个空值
df.describe()  ### 获取 df 的统计信息
            sale_amt
count   198.000000
mean   2765.545152
```

```
std      709.557639
min       22.000000
25%     2452.725000
50%     2655.850000
75%     3023.500000
max     9106.440000
# 建立图像
plt.figure()
# 画箱线图
bp = df.boxplot()
# flies 为异常值的标签
x = bp['fliers'][0].get_xdata()
y = bp['fliers'][0].get_ydata()
y.sort()

# 用 annotate 添加注释
for i in range(len(x)):
    plt.annotate(y[i], xy = (x[i],y[i]), xytext=(x[i]+0.1-0.8/(y[i]-y[i-1]),y[i]))

plt.show()
```

运行结果如图 2-5 所示。

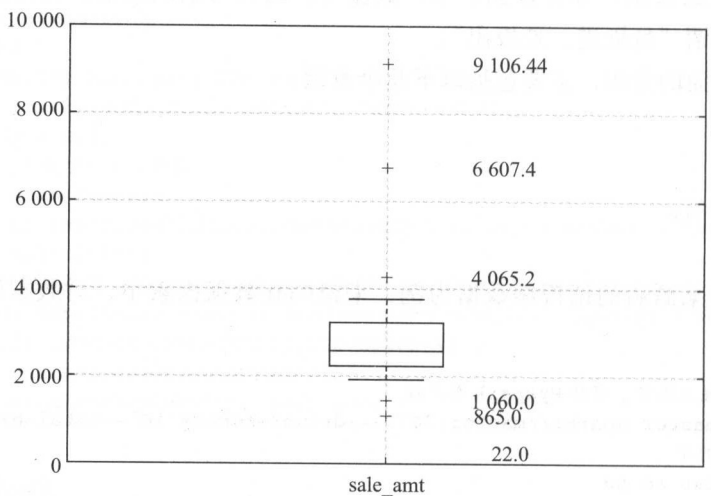


图 2-5 销售额箱线图检测异常值

从以上分析可知，销售额列存在两个空值、六个可能的异常值，其中 865.0 和 1060.0 有可能属于正常值，当然也需要和相关业务员沟通，对于其他数值，需要进一步分析异常值产生的原因，然后确定数据的去留。

2.4.3 数据特征分析

对数据质量有基本了解后，接下来就可以对数据的特征进行分析，数据特征分析一般

包括以下内容：

- 特征分布分析；
- 对比分析；
- 统计量分析。

特征一般指用于模型训练的变量，原始数据中的特征有些是数值，有些是字符或其他格式信息，但在进行机器学习前，都需要转换为数值。根据实际情况，有时需要根据已有特征生成或衍生出新特征，如根据用户年龄衍生出表示老、中、青的新特征；有时需要对一些特征进行规范化、标准化等转换，尤其是回归类模型。

1. 数据特征分析实例

特征的分布分析有助于发现相关数据的分布特征、分布类型、分布是否对称等，可以使用数据可视化方法，这样便于直观发现特征的异常值。下面以用户信息数据为例，分析用户的年龄特征、职业特征等。

```
from pyspark.sql import SparkSession
from pyspark.sql import Row

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

sc = spark.sparkContext
# 加载 textfile 文件并转换为行式
userrdd = sc.textFile("hdfs://master:9000/u01/bigdata/data/u.user").map(lambda
          line: line.split("|"))
# 利用反射机制把 RDD 转换为 DataFrame
df = userrdd.map(lambda fields: Row(name=fields[0], age=int(fields[1]),gender=fi
     elds[2],occupation=fields[3],zip=fields[4]))

# 把 dataframe 注册为一个 table
schemauser = spark.createDataFrame(df)
schemauser.createOrReplaceTempView("user")

# 在 table 上运行 SQL
age = spark.sql("SELECT age FROM user")
# 把运行结果转换为 RDD
ages = age.rdd.map(lambda p: p.age).collect()
hist(ages, bins=20, color='lightblue', normed=True)
```

运行结果如图 2-6 所示。

从图 2-6 可以看出，最小年龄在 10 岁左右，最大年龄超过 70 岁，大部分是 20 ～ 40 岁。

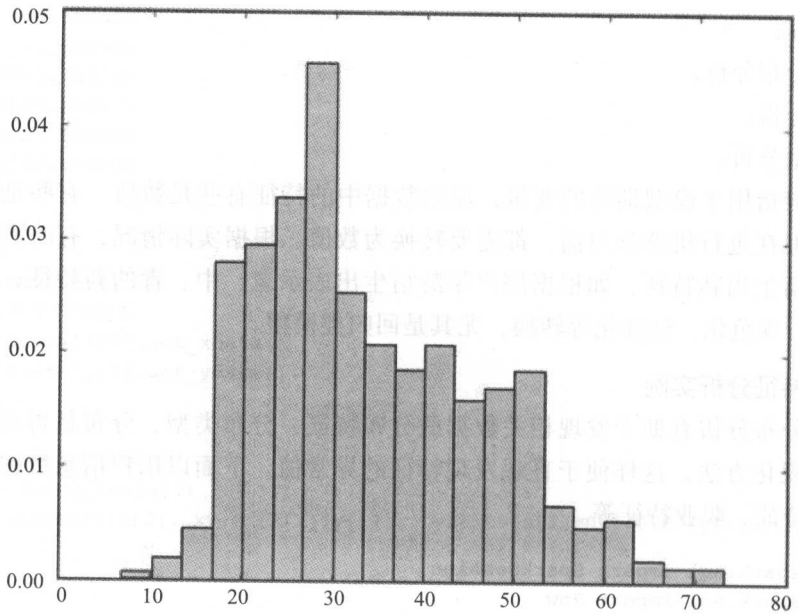


图 2-6 用户年龄特征分布图

我们还可以进一步分析用户职业分布特征。

```
# 选取用户职业数据
count_occp = spark.sql("SELECT occupation,count(occupation) as cnt FROM user
          Group by occupation order by cnt")
# 查看前 5 行数据
count_occp.show(5)
+----------+---+
|occupation|cnt|
+----------+---+
| homemaker|  7|
|    doctor|  7|
|      none|  9|
|    lawyer| 12|
|  salesman| 12|
+----------+---+
# 获取职业名称及职业数，以便画出各职业总数图形
# 把运行结果转换为 RDD
x_axis = count_occp.rdd.map(lambda p: p.occupation).collect()
y_axis = count_occp.rdd.map(lambda p: p.cnt).collect()

pos = np.arange(len(x_axis))
width = 1.0
### 隐式新增一个 figure，或为当前 figure 新增一个 axes
ax = plt.axes()
ax.set_xticks(pos + (width / 2))    ### 设置 x 轴刻度
ax.set_xticklabels(x_axis)          #### 为对应刻度打上标签
```

```
plt.bar(pos, y_axis, width, color='orange')
plt.xticks(rotation=30)          ####x 轴上的标签旋转 30 度
fig = matplotlib.pyplot.gcf()    ### 获取当前 figure 的应用
fig.set_size_inches(16, 10)      ### 设置当前 figure 大小
```

从图 2-7 所示的用户职业分布图可以看出，学生占绝大多数，其次是其他职业从业者、教育工作者、管理者、工程师等。医生、家庭主妇或许平时较忙，故数量比较少。

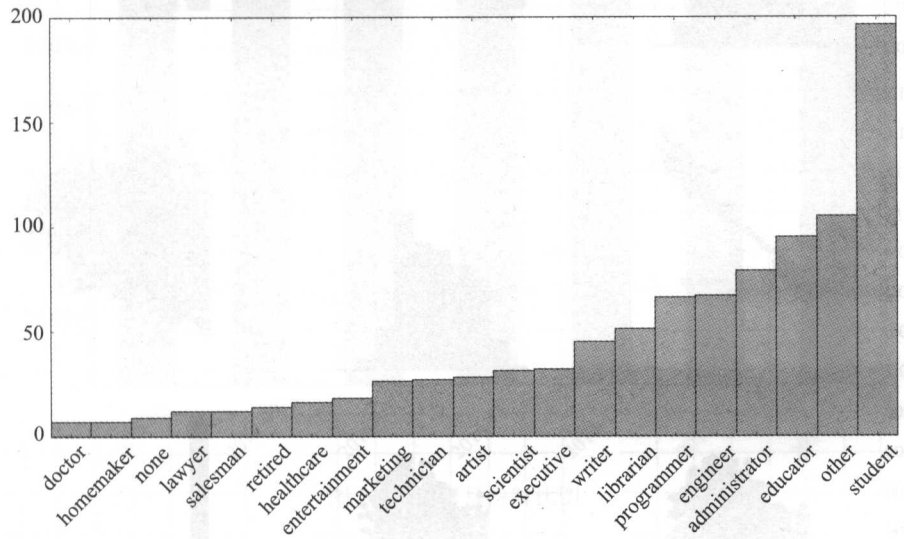


图 2-7　用户职业分布图

2. 特征分布及相关性分析

在数据探索阶段分析特征分布、特征间的相关性等，将为后续的特征选择、特征提取将提供重要依据。以下是对共享单车数据的特征分析，详细内容可参考 9.3 节。

```
### 探索特征间分布、相关性等
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt


data_pd=data1.toPandas()
sns.set(style='whitegrid',context='notebook')
cols=['temp','atemp','label']
sns.pairplot(data_pd[cols],size=2.5)
plt.show()
```

运行结果如图 2-8 所示。

3. 对比分析

下面以销售数据为例进行对比分析，运行结果如图 2-9 所示。

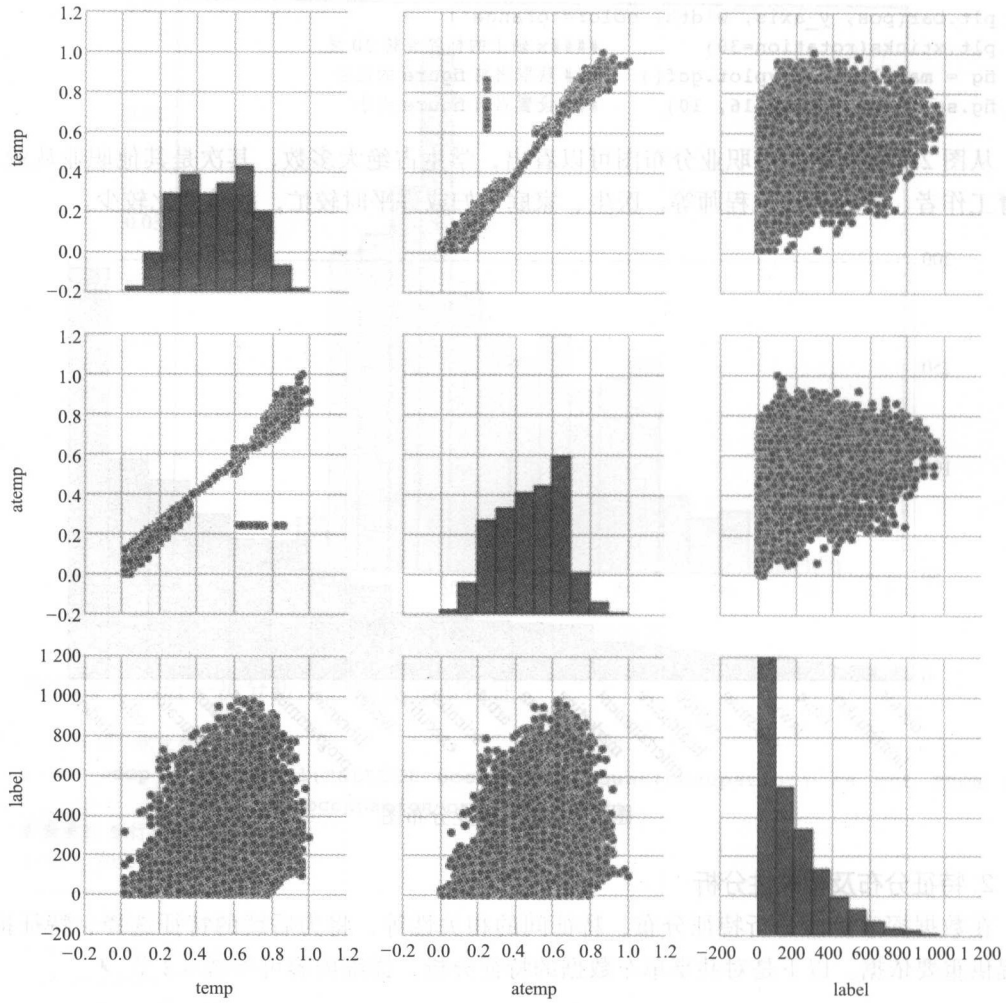


图 2-8 hours 数据集特征分布及相关性示例图

```
### 导入需要的库
import pandas as pd
### 把日期列作为索引，并转换为日期格式
df = pd.read_csv("/home/hadoop/data/catering_sale.csv",header=0,index_col='sale_
     date',parse_dates=True)
### 把空值置为 0
df1=df.fillna(0)
### 根据年月求和
df_ym=df1.resample('M',how='sum')
## 取年月
df2=df_ym.to_period('M')
## 数据可视化
df2.plot(kind='bar',rot=30)
```

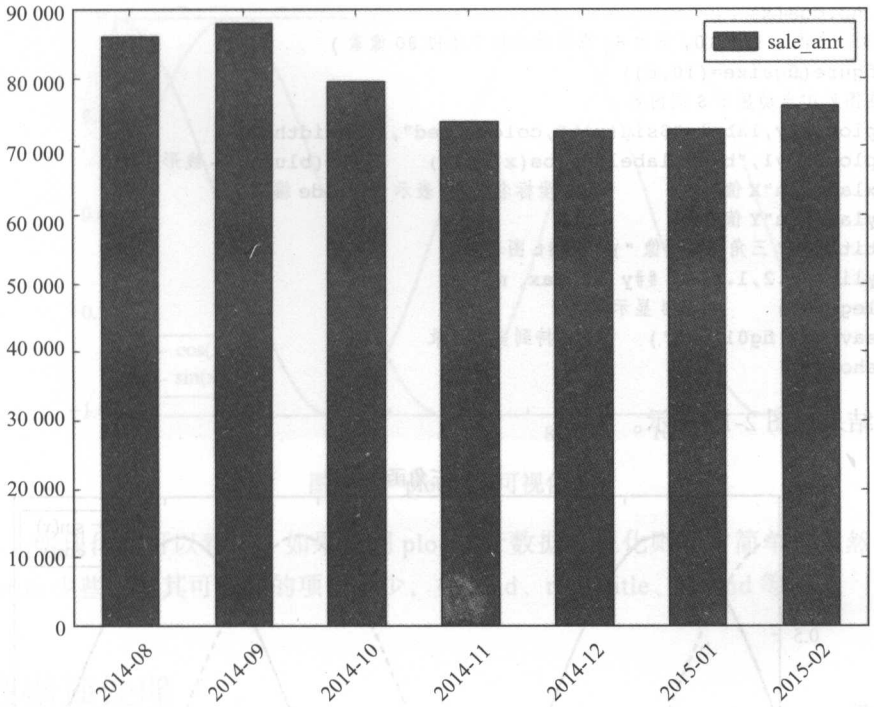


图 2-9　销售月份对比图

2.4.4　数据的可视化

数据的可视化是数据探索、数据分析中的重要任务，通过可视化可以帮助我们发现数据的异常值、特征的分布情况等，为数据预处理提供重要支持。Spark 目前对数据的可视化功能还很弱或还没有，不过没关系，我们可以借助 Python 或 R 等可视化功能，Python 和 R 在数据可视化方面功能很强大，这里以 Python 的数据可视化为例。Python 的数据表现能力很强，其可视化可以使用 matplotlib 或 plot 等方法。matplotlib 是一种比较低级但强大的绘图工具，可以进行很多定制化，但往往需要较大代码来实现；plot 是一种非常简洁的绘图工具，它主要基于 pandas 基础之上。接下来我们通过两个示例来具体说明。

下例是通过 matplotlib 可视化 sin(x) 和 $\cos(x^2)$ 函数的图形。

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib
import matplotlib.pyplot as plt

plt.rcParams['font.sans-serif']=['SimHei']    ### 显示中文
plt.rcParams['axes.unicode_minus']=False  ## 防止坐标轴上的 "-" 号变为方块
x = np.linspace(0, 10, 100)
y = np.sin(x)
```

```
y1 = np.cos(x)
## 绘制一个图，长为 10，宽为 6（默认值是每个单位 80 像素）
plt.figure(figsize=(10,6))
### 在图列中自动显示 $ 间内容
plt.plot(x,y,label="$sin(x)$",color="red",linewidth=2)
plt.plot(x,y1,"b--",label="$cos(x^2)$")     ###b(blue),-- 线形
plt.xlabel(u"X 值")          ##X 坐标名称，u 表示 unicode 编码
plt.ylabel(u"Y 值")
plt.title(u" 三角函数图像 ")    ##t 图名称
plt.ylim(-1.2,1.2)    ##y 上的 max、min 值
plt.legend()          ## 显示图例
plt.savefig('fig01.png')    ## 保持到当前目录
plt.show()
```

运行结果如图 2-10 所示。

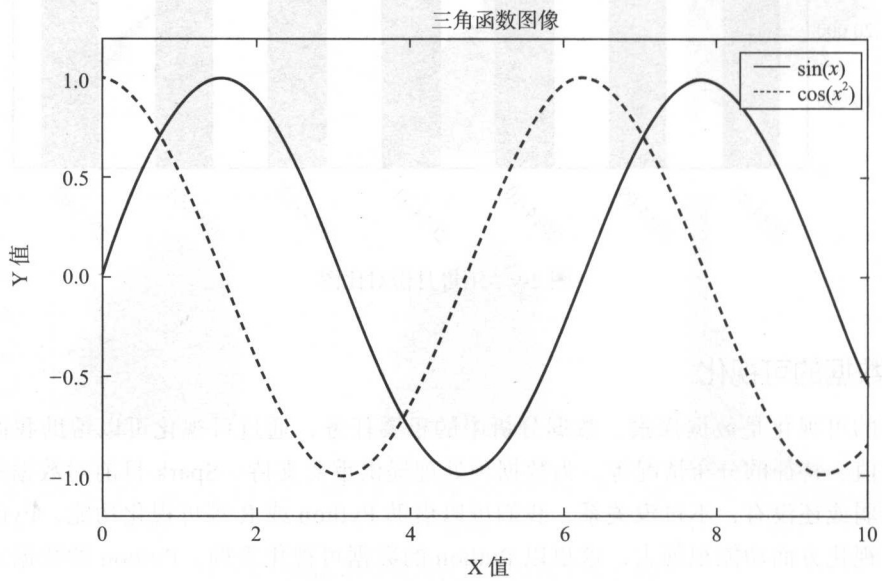


图 2-10　matplot 数据可视化

同样的，如果我们使用 plot 对这些数据来进行可视化，代码可以非常简洁，但定制化方面可能要弱一些。

```
from pandas import DataFrame
import pandas as pd
import numpy as np

x = np.linspace(0, 10, 100)
df=DataFrame({'sin(x)':np.sin(x),'cos(x)':np.cos(x)},index=x)
df.plot()
```

显示图形如图 2-11 所示。

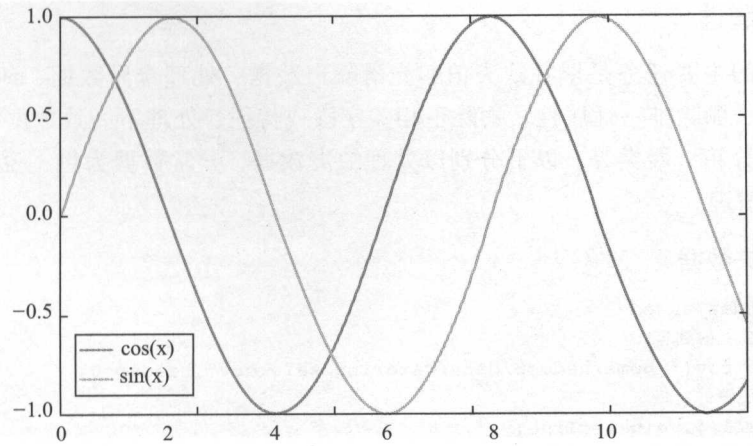


图 2-11　plot 数据可视化

从以上实现代码可以看出，如果使用 plot 进行数据可视化则非常简单，虽然定制化要比 matplotlib 少些，但其可定制的项也不少，如 kind、rot、title、legend 等。

2.5　数据预处理

前面我们介绍了探索数据的一些方法，通过对数据的探索，可以帮助我们发现一些奇异数据、缺失数据、特征的类别及其分布情况等信息。而这些信息正是对数据预处理的重要依据。在数据分析、机器学习中，数据的预处理非常关键，尤其是涉及大数据的处理，往往是比较费时、费神的过程，有时还需要往返多次。当然，如果数据预处理做得好，除提高数据质量外，更能极大提高模型的性能。

数据的预处理一般包括数据清理、数据变换、数据集成、数据归约等，如图 2-12 所示。

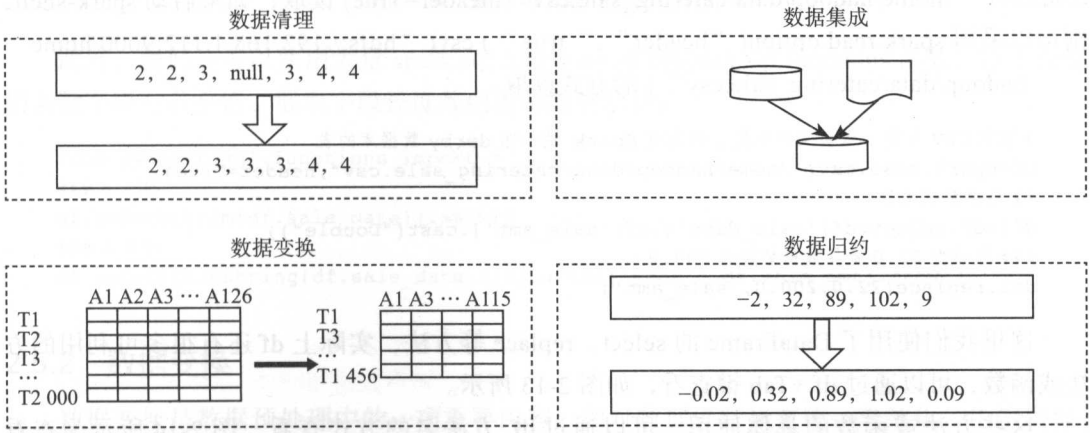


图 2-12　数据预处理示意图

2.5.1 数据清理

数据清理的主要任务是填补缺失值、光滑噪声数据、处理奇异数据、纠正错误数据、删除重复数据、删除唯一性属性、去除不相关字段或特征、处理不一致数据等。噪声数据的处理方法为分箱、聚类等。以下分别以处理缺失数据、异常数据为例，说明在 Spark 中如何进行数据清理。

1. 处理缺失数据

```
import pandas as pd
## 读取 HDFS 上的数据
df=pd.read_csv("/home/hadoop/data/catering_sale.csv",header=0)
## 定位数据集中的空值
df[df.isnull().values==True]
## 显示结果如下，说明有 2 个空值
    sale_date  sale_amt
13  2015/2/14       NaN
32  2015/1/26       NaN
### 以 0 填补空值
df.fillna(0)
## 或该列的平均值填补空值
df['sale_amt'].fillna(df['sale_amt'].count())
## 或用该列前一行值填补空值
df.fillna(method='pad')
```

2. 处理奇异数据

在数据探索阶段，我们发现销售数据文件 catering_sale.csv 中有 6 个可能的奇异数据，假设与相关人员核实后，只有 22.0 为奇异数据或错误数据。对错误数据我们一般采用删除或替换的方法，这里我们采用 Spark SQL 来处理奇异数据。

首先把数据复制到 HDFS，用 Spark 读取数据，如果启动 Pyspark，则可以通过 spark.read.csv(“/home/hadoop/data/catering_sale.csv”,header=True) 读取；如果启动 spark-shell，则可以采用 spark.read.option(“header”,”true”).csv(“hdfs://192.168.1.112:9000/home/hadoop/data/catering_sale.csv”) 的方式读取。

```
# 读取 CSV 文件，保留文件标题，并创建 Spark 的一张 derby 数据库的表
df=spark.read.csv("/home/hadoop/data/catering_sale.csv",header=True)
## 转换数据类型
df1=df.select(df['sale_date'],df['sale_amt'].cast("Double"))
### 假设把 22.0 奇异数据替换为 200.0
df1.replace(22.0,200.0,'sale_amt')
```

这里我们使用了 DataFrame 的 select、replace 等方法，实际上 df 还有很多可利用的方法或函数，可以通过 df.+Tab 键查看，如图 2-13 所示。

这些方法或函数的具体使用，可以通过 df. 方法名的方式查看，图 2-14 所示为查看 df.filter 的详细用法。

```
In [2]: df.
df.agg                          df.drop                 df.join                 df.sortWithinPartitions
df.alias                        df.drop_duplicates      df.limit                df.sql_ctx
df.approxQuantile               df.dropDuplicates       df.na                   df.stat
df.cache                        df.dropna               df.orderBy              df.storageLevel
df.checkpoint                   df.dtypes               df.persist              df.subtract
df.coalesce                     df.explain              df.printSchema          df.take
df.collect                      df.fillna               df.randomSplit          df.toDF
df.columns                      df.filter               df.rdd                  df.toJSON
df.corr                         df.first                df.registerTempTable    df.toLocalIterator
df.count                        df.foreach              df.repartition          df.toPandas
df.cov                          df.foreachPartition     df.replace              df.union
df.createGlobalTempView         df.freqItems            df.rollup               df.unionAll
df.createOrReplaceTempView      df.groupby              df.sample               df.unpersist
df.createTempView               df.groupBy              df.sampleBy             df.where
df.crossJoin                    df.head                 df.schema               df.withColumn
df.crosstab                     df.intersect            df.select               df.withColumnRenamed
df.cube                         df.is_cached            df.selectExpr           df.withWatermark
df.describe                     df.isLocal              df.show                 df.write
df.distinct                     df.isStreaming          df.sort                 df.writeStream
```

图 2-13　其他可利用的方法或函数

```
In [3]: df.filter?
Signature: df.filter(condition)
Docstring:
Filters rows using the given condition.

:func:`where` is an alias for :func:`filter`.

:param condition: a :class:`Column` of :class:`types.BooleanType`
    or a string of SQL expression.

>>> df.filter(df.age > 3).collect()
[Row(age=5, name=u'Bob')]
>>> df.where(df.age == 2).collect()
[Row(age=2, name=u'Alice')]

>>> df.filter("age > 3").collect()
[Row(age=5, name=u'Bob')]
>>> df.where("age = 2").collect()
[Row(age=2, name=u'Alice')]
```

图 2-14　查看 df.filter 的使用方法

此外，我们还可以使用大量 spark.sql.functions 或 pyspark.sql.functions 函数，以下是使用去除字段左右空格、截取字段长度等内置函数的示例：

```
from pyspark.sql.functions import *
### 去空格
df.select(trim(df.sale_date)).show()
### 去年份
df.select(substring(df.sale_date,1,4).alias('year'),df.sale_amt).show()
```

2.5.2　数据变换

数据变换是数据预处理中的一项重要内容，如对数据进行规范化、离散化、衍生指标、类别特征数值化、平滑噪声等都属于数据变换。在 Spark ML 中有很多现成的数据变换算

法，利用这些算法可极大提高整个数据处理的效率，表 2-2 所示只是概况，更多详细信息请参考第 4 章。

表 2-2 Spark ML 自带的数据变换算法

数据预处理	算法	功能简介
特征抽取	TF-IDF	统计文档的词频—逆向文件频率（TF-IDF）
	Word2Vec	将文档转换为向量
特征转换	Tokenization	Tokenization 将文本划分为独立个体
	StopWordsRemover	删除了所有停用词
	PCA	使用 PCA 方法可以对变量集合进行降维
	StringIndexer	StringIndexer 将字符串列编码为标签索引列
	OneHotEncoder	将标签指标映射为二值（0 或 1）向量
	Normalizer	规范每个向量以具有单位范数
	StandardScaler	标准化每个特征使得其有统一的标准差以及（或者）均值为 0，方差为 1
	VectorAssembler	将给定的多列表组合成一个单一的向量列

这里我们以卡方检验为例，分析如何根据特征的贡献率来选择特征。假设我们有很多特征，如表示时间特征的季节（season）、年月（yr）、月份（mnth）、是否节假日（holiday）、是否周末（weekday），表示天气的特征 weathersit、temp 等。为了使用卡方检验来选择这些特征，首先需要把各特征组合为一个特征向量，然后，把整合后的特征向量及选择特征个数等代入卡方模型中，详细代码如下：

```
// 定义特征向量
featuresArray =["season","yr","mnth","hr","holiday","weekday","workingday",
"weathersit","temp","atemp","hum","windspeed"]

### 把各特征组合成特征向量 features
assembler = VectorAssembler(inputCols=featuresArray,outputCol="features")
### 选择贡献度较大的前 5 个特征
selectorfeature = ChiSqSelector(numTopFeatures=5, featuresCol="features",outputCol
="selectedFeatures", labelCol="label")
```

2.5.3 数据集成

数据集成是将多文件或者多数据库中的数据进行合并，然后存放在一个一致的数据存储中。数据集成一般通过 join、union 或 merge 等关键字把两个（或多个）数据集连接在一起，Spark SQL（包括 DataFrame）有 join 方法，Pandas 下有 merge 方法。数据集成往往需要耗费很多资源，尤其是大数据间的集成涉及 shuffle 过程，有时需要牵涉多个节点，所以除了数据一致性外，性能问题常常不请自来，需要我们特别留心。

传统数据库一般是在单机上采用 hash join 方法，如果在分布式环境中，采用 join 时，可以考虑充分利用分布式资源进行平行化。当然，在进行 join 之前，对数据过滤或归约也是常用的优化方法。

Spark SQL 中有三种 join 方法：

- broadcast hash join：如果 join 的表中有一张大表和一张较小的表，可以考虑把小表广播分发到大表所在的分区节点上，分别并发地与其上的分区记录进行 hash join。
- shuffle hash join：如果两张表都不小，对数据量较大的表进行广播分发就不太适合。这种情况下，可以根据 join key 相同必然分区相同的原理，将两张表分别按照 join key 进行重新组织分区，这样就可以将 join 分而治之，划分为很多小 join，充分利用集群资源并行化。
- sort merge join：对数据量较大的表也可以考虑使用 sort merge join 方法，先将两张大表根据 join key 进行重新分区，两张表数据会分布到整个集群，以便分布式并行处理，然后，对单个分区节点的两表数据分别进行排序，最后，对排好序的两张分区表数据执行 join 操作。

DataFrame 中 join 的方式有（或 merge）内连接、左连接、右连接等。

2.5.4　数据归约

大数据是机器学习的基础，但大数据往往数据量非常大，有时我们可以通过数据归约技术删除或减少冗余属性（或维）、精简数据集等，使归约后的数据比原数据小，甚至小很多，但仍然接近于保持原数据的完整性，且结果与归约前结果相同或几乎相同。表 2-3 所示列举了 Spark ML 自带的特征选择或降维算法。

表 2-3　Spark ML 自带的数据选择算法及功能简介

数据预处理	算法	功能简介
特征选择或降维	VectorSlicer	得到一个新的原始特征子集的特征向量
	RFormula	通过 R 模型公式来将数据中的字段转换为特征值
	PCA	使用 PCA 方法可以对变量集合进行降维
	SVD	$Am \times n \approx Um \times r \Sigma r \times rVr \times n$（r<<m，r<<n）
	ChiSqSelector	根据分类的卡方独立性检验来对特征排序，选取类别标签主要依赖的特征

选择特征或降维是机器学习中重要的处理方法，我们可以使用上述这些方法在减少特征个数、消除噪声等问题的同时，维持原始数据的内在结构或主要特征。尤其是降维，在大数据、机器学习中具有重要作用，以下通过两个实例说明 SVD、PCA 算法的具体使用（目前 Spark MLlib 支持 SVD 及 PCA）。

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.SingularValueDecomposition
import org.apache.spark.mllib.linalg.Vector
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.distributed.RowMatrix

val data = Array(
Vectors.dense(1,2,3,4,5,6,7,8,9),
```

```
Vectors.dense(5,6,7,8,9,0,8,6,7),
Vectors.dense(9,0,8,7,1,4,3,2,1),
Vectors.dense(6,4,2,1,3,4,2,1,5),
Vectors.dense(4,5,7,1,4,0,2,1,8))

val dataRDD = sc.parallelize(data, 2)
val mat: RowMatrix = new RowMatrix(dataRDD)

//保留前3个奇异值，需要获得U成员
val svd = mat.computeSVD(3, computeU = true)
//通过访问svd对象的V、s、U成员分别拿到进行SVD分解后的右奇异矩阵、奇异值向量和左奇异矩阵
val U: RowMatrix = svd.U  //左奇异矩阵
val s: Vector = svd.s  //从大到小的奇异值向量
[30.88197557931219,10.848035248251415,8.201924156089822]
val V: Matrix = svd.V //右奇异矩阵
-0.33309047675110115  0.6307611082680837    0.10881297540284612
-0.252559026169606    -0.13320654554805747  0.4862541277385016
-0.3913180354223819   0.3985110846022322    0.20656596253983592
-0.33266751598925126  0.25621153877501424   -0.3575093420454635
-0.35120996186827147  -0.24679309180949208  0.16775460006130793
-0.1811460330545444   0.03808707142157401   -0.46853660508460787
-0.35275045425261     -0.19100365291846758  -0.26646095393100677
-0.2938422406906167   -0.30376401501983874  -0.4274842789454556
-0.44105410502598985  -0.4108875465911952   0.2825275707788212
```

同样是这个矩阵 data，以下我们用 PCA 进行分解，看一下效果及与 SVD 的异同。SVD 分解后右奇异矩阵 V 与 PCA 降维后的矩阵 pc 很相似。

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.linalg.distributed.RowMatrix

val data = Array(
Vectors.dense(1,2,3,4,5,6,7,8,9),
Vectors.dense(5,6,7,8,9,0,8,6,7),
Vectors.dense(9,0,8,7,1,4,3,2,1),
Vectors.dense(6,4,2,1,3,4,2,1,5),
Vectors.dense(4,5,7,1,4,0,2,1,8))

val dataRDD = sc.parallelize(data, 2)
val mat: RowMatrix = new RowMatrix(dataRDD)
val pc: Matrix = mat.computePrincipalComponents(3)
-0.3948204553820511   -0.3255749878678745   0.1057375753926894
0.1967741975874508    0.12066915005125914   0.4698636365472036
-0.09206257474269655  -0.407047128194367    0.3210095555021759
0.12315980051885281   -0.6783914405694824   -0.10049065563002131
0.43871546256175087   -0.12704705411702932  0.2775911848440697
-0.05209780173017968  0.10583033338605327   -0.6473697692806737
0.422474587406277     -0.27600606797384     -0.13909137208338707
```

```
0.46536643478632944   -0.172268807944553    -0.349731653791416
0.4376262507870099    0.3469015236606571    0.13076351966313637
```

使用 PCA 降维，利用 Pyspark 的画图功能，可以对新生成的特征的方差贡献度进行可视化，图 2-15 所示为对 hour.csv 数据通过 PCA 处理后，重要特征的排序情况。

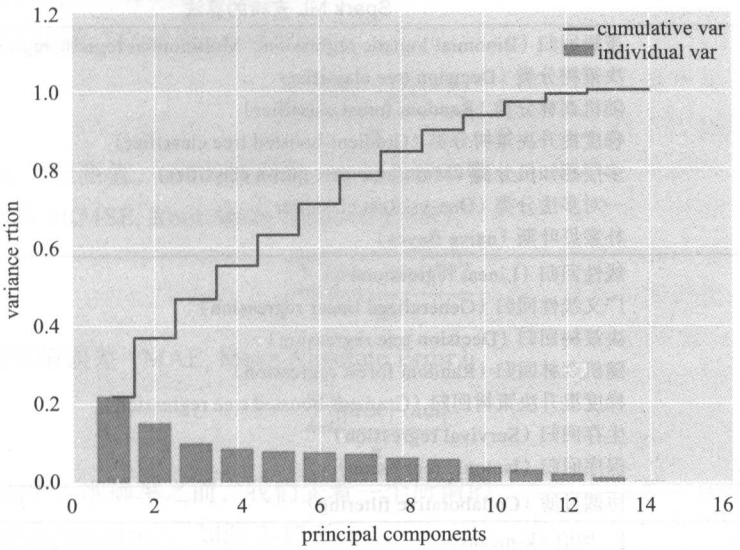


图 2-15　hour.csv 数据的 PCA 分析图

2.6　构建模型

前面我们介绍了准备阶段，包括加载数据、探索数据、预处理数据等，数据准备阶段往往是最费时间和精力的，这个问题解决了，常常又会出现新问题，需要返回多次。一般而言，数据准备阶段从时间上来说可能要占据 60% 左右，有时更多。

数据准备好以后，接下来就是构建模型。模型是机器学习、数据挖掘等的核心，构建模型涉及确定模型或算法、设置参数、训练模型等，其大致流程如图 2-16 所示。

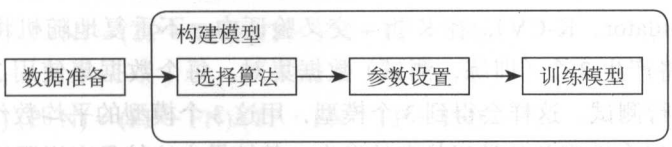


图 2-16　构建模型流程

选择算法主要依据业务需求、数据特征等，Spark 目前支持分类、回归、推荐等这些常用而且重要的算法，具体如表 2-4 所示。如何选择算法，需要考虑业务需求、数据特征、算法适应性、个人经验等，当然，也可选择几种方法，然后进行比较，或采用集成学习的

方式。复合多种算法也是选项之一，如先采用聚类方法对数据进行聚类，然后对不同类别的数据进行预测或推荐，有时会得到更好的结果。如果你觉得选择比较难或还不好确定，可以先从简单或熟悉的算法开始，然后，不断完善和优化。

表 2-4 Spark ML 目前支持的算法

类型	Spark ML 支持的算法
分类	逻辑回归（Binomial logistic regression、Multinomial logistic regression） 决策树分类（Decision tree classifier） 随机森林分类（Random forest classifier） 梯度提升决策树分类（Gradient-boosted tree classifier） 多层感知机分类（Multilayer perceptron classifier） 一对多法分类（One-vs-Rest classifier ） 朴素贝叶斯（naive Bayes）
回归	线性回归（Linear regression） 广义线性回归（Generalized linear regression） 决策树回归（Decision tree regression） 随机森林回归（Random forest regression） 梯度提升决策树回归（Gradient-boosted tree regression） 生存回归（Survival regression） 保序回归（Isotonic regression）
推荐	协调过滤（Collaborative filtering）
聚类	K- 均值（k-means） 高斯混合（Gaussian Mixture Model） 主题模型（latent Dirichlet allocation（LDA）） 二分 K 均值（bisecting k-means）

确定算法后，一般还需要设置一些参数，如训练决策树时需要选择迭代次数、纯度计算方法、树的最大高度等，此外，对准备好的数据需要进行划分，一般划分为训练数据和测试数据，有的会把训练数据进一步划分为训练数据集、验证数据集。Spark 提供多种随机划分数据的方法，如 randomSplit、CrossValidator 等。这些方法的使用在 2.9 节中会具体说明。

训练数据用于训练模型，测试数据用于验证模型，因这个环节的验证是在模型训练过程实现的，所以它一般也认为隶属于模型建立过程。这种划分数据进行验证的方法一般称为交叉验证（CrossValidator，CV），有些交叉验证把数据分成 K 组，如 K 折交叉验证（K-fold Cross Validator，K-CV），在 K 折 – 交叉验证中，不重复地随机将数据集划分为 K 对，如 K=3，则将产生 3 个（训练，测试）数据集对，每个数据集使用 2/3 的数据进行训练，1/3 的数据进行测试。这样会得到 3 个模型，用这 3 个模型的平均数作为最终模型的性能指标。K-CV 可以有效避免欠学习状态的发生，其结果也比较具有说服性。

2.7 模型评估

模型构建以后，接下来就需要对模型的性能、与目标的切合度等进行一些评估，模型

评估是模型开发过程中不可或缺的一部分。在构建模型的过程中，会产生一些评估指标，如精确度、ROC、RMSE 等，这些指标是重要而且基础的，但不是唯一和最终指标，除了这些指标外，我们还应该评估模型对业务的提示或商业目标的达成等方面的贡献。一个好的模型不但要有好的技术指标，更要为解决实际问题提供帮助，有时后者显得更为重要。

Spark 中常用的几个评估算法如下。

1）均方差（MSE, Mean Squared Error）：

$$\frac{\sum(\text{prec}-\text{act})**2}{n}$$

式中：prec 为预测值，act 为实际值，n 为总样本数。

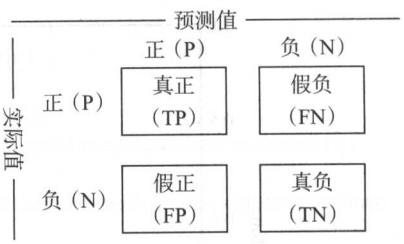
2）均方根差（RMSE, Root Mean Squared Error）：

$$\sqrt{\frac{\sum(\text{pec}-\text{act})**2}{n}}$$

3）平均绝对值误差（MAE, Mean Absolute Error）：

$$\frac{\sum|\text{prec}-\text{act}|}{n}$$

在了解正确率、准确率之前，我们先看一个所谓的混淆矩阵（confusion matrix），如图 2-17 所示。



2-17　混淆矩阵

混淆矩阵为是一个简单矩阵，用于展示一个二分类器的预测结果，其中 T 为 True、F 为 False、N 为 Negative、P 为 Postitive。

- 真正（TP）：被模型预测为正的正样本数；可以称作判断为真的正确率；
- 真负（TN）：被模型预测为负的负样本数；可以称作判断为假的正确率；
- 假正（FP）：被模型预测为正的负样本数；可以称作误报率；
- 假负（FN）：被模型预测为负的正样本数；可以称作漏报率。

正确率（Accuracy），反映了分类器对整个样本的判定能力——能将正的判定为正，负的判定为负。

$A = (\text{TP} + \text{TN})/(\text{P}+\text{N}) = (\text{TP} + \text{TN})/(\text{TP} + \text{FN} + \text{FP} + \text{TN})$

错误率（Error）：

$E = (\text{FP} + \text{FN})/(\text{P}+\text{N}) = (\text{FP} + \text{FN})/(\text{TP} + \text{FN} + \text{FP} + \text{TN})$

准确率（Precision），反映了被分类器判定的正例中真正的正例样本的比重：

$P = \text{TP}/(\text{TP}+\text{FP})$

召回率 (Recall)，反映了被正确判定的正例占总的正例的比重。

$R = \text{TP}/(\text{TP}+\text{FN}) = 1-\text{FN}/T$;

F1-Measure：

$F1 = 2P*R/(P+R)$

真阳性率 (TPR) 代表分类器预测的正类中实际正实例占所有正实例的比例：

TPR= TP/(TP+FN)

假阳性率 (FPR) 代表分类器预测的正类中实际负实例占所有负实例的比例：

FPR= FP/(FP+TN)

以上这些都属于静态的指标，当正负样本不平衡时会存在严重的问题。极端情况下，比如正负样本比例为 1:99（有些领域并不少见），那么一个分类器只要把所有样本都判为负，它就拥有了 99% 的精确度，但这时的评价指标是不具有参考价值的。另外，很多分类器都不是简单地给出一个正或负（0 或 1）的分类判定，而是给出一个分类的倾向程度，比如贝叶斯分类器输出的分类概率。对于这些分类器，当你取不同阈值时，就可以得到不同的分类结果及分类器评价指标，依此人们又发明出 ROC 曲线以及 AUC（ROC 曲线包围面积）指标来衡量分类器的总体可信度。ROC 曲线将 FPR 和 TPR 定义为 x 和 y 轴，这样就描述了真阳性和假阳性在不同决策阈值下的关系。AUC 越大说明模型性能越好，ROC 曲线如图 2-18 所示。

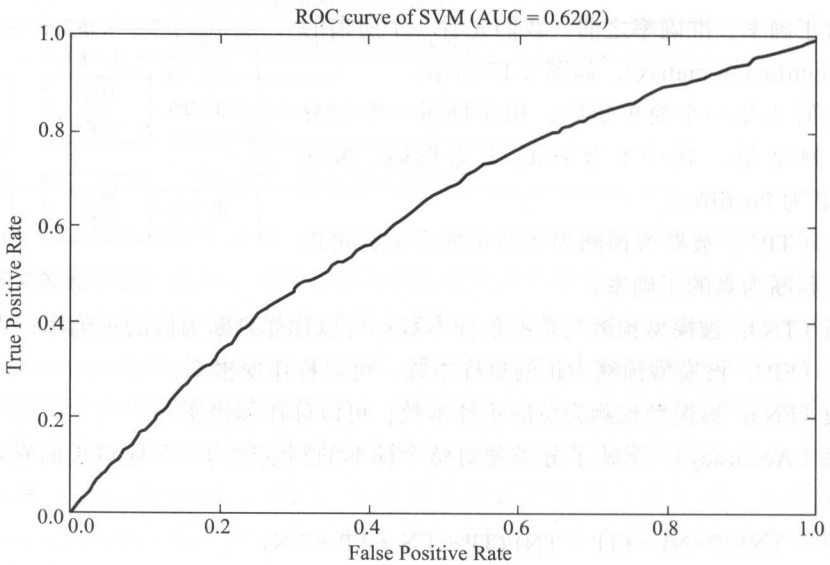


图 2-18 ROC 曲线示意图

下面通过一个实例说明 Spark 一些评估指标的使用：

```
import org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.classification.LogisticRegressionModel
import org.apache.spark.sql.SparkSession
```

```
import org.apache.spark.sql.functions._
import org.apache.spark.mllib.evaluation.RegressionMetrics

val path="file:///u01/bigdata/spark/data/mllib/sample_libsvm_data.txt"
val data=spark.read.format("libsvm").load(path)
val Array(trainingData, testData) = data.randomSplit(Array(0.7, 0.3), seed
= 1234L)
//参数说明
// threshold 变量用来控制分类的阈值，默认值为 0.5
val lr = new LogisticRegression()
.setThreshold(0.6).setMaxIter(10).setRegParam(0.3).setElasticNetParam(0.8)
val lrModel = lr.fit(trainingData)

val predictions = lrModel.transform(testData)
predictions.show()

//计算 MSE、MAE、RMSE 等
val evaluator = new BinaryClassificationEvaluator()
        .setLabelCol("label")
val accuracy = evaluator.evaluate(predictions)
val rm2 = new RegressionMetrics(predictions.select("prediction", "label").rdd.
map(x =>(x(0).asInstanceOf[Double], x(1).asInstanceOf[Double])))
println("MSE: " + rm2.meanSquaredError)
println("MAE: " + rm2.meanAbsoluteError)
println("RMSE Squared: " + rm2.rootMeanSquaredError)

//将其作为多分类结果进行评估，可计算 F1、准确率、召回率、正确率
val multiclassClassificationEvaluator = new MulticlassClassificationEvaluator()
def printlnMetric(metricName: String): Unit = {
    println(metricName + " = " + multiclassClassificationEvaluator.setMetricName
    (metricName).evaluate(predictions))
}

printlnMetric("f1")//f1 = 0.9646258503401359
printlnMetric("weightedPrecision")//weightedPrecision = 0.9675324675324675
printlnMetric("weightedRecall")//weightedRecall = 0.9642857142857142
printlnMetric("accuracy")//accuracy = 0.9642857142857143

//将其作为二分类结果进行评估，可计算 areaUnderROC、areaUnderPR
val binaryClassificationEvaluator = new BinaryClassificationEvaluator()
def printlnMetric(metricName: String): Unit = {
    println(metricName + " = " + binaryClassificationEvaluator.setMetricName
    (metricName).evaluate(predictions))
}

printlnMetric("areaUnderROC") //结果为 areaUnderROC = 0.9944444444444444
printlnMetric("areaUnderPR")//结果为 areaUnderPR = 0.9969948018193632
//分类正确且分类为 1 的样本数量 TP 是 17
predictions.filter($"label" === $"prediction").filter($"label"===1).count
//分类正确且分类为 0 的样本数量 TN 是 10
```

```
predictions.filter($"label" === $"prediction").filter($"label"===0).count
// 分类错误且分类为 0 的样本数量 FN 是 1
predictions.filter($"label" !== $"prediction").filter($"prediction"===0).count
// 分类错误且分类为 1 的样本数量 FP 是 0
predictions.filter($"label" !== $"prediction").filter($"prediction"===1).count
```

运行后得出：

准确率：TP/(TP+FP)=17/(17+0)=1

召回率：TP/(TP+FN) = 17/(17+1)=0.944444

2.8 组装

前面几节我们对数据集进行了探索，之后进行大量的数据清理、转换等数据预处理工作，接着进行构建模型、评估模型。评估模型前我们需要将数据集随机划分为训练集和测试集。假如数据有变化，如新增数据，如何保证训练集和测试集上的操作保持一致？如果数据清理、数据转换等有很多步骤，如何保证这些步骤依次执行？

采用 Spark Pipeline 能很好解决这些问题。也就是说，我们只要把这些任务作为 Pipeline 的 Stage，按照其本身的执行次序把这些 stage 组装到一个 Pipeline 上即可。当然，如果任务比较复杂，我们也可以采用多个 Pipeline，然后把这些 Pipeline 组装到一个新的 Pipeline 中。

组装的一般步骤如下：

1）创建 Pipeline，将各个 Stage 依次组装在一起，如：

```
val pipeline = new Pipeline()
    .setStages(Array(tokenizer, hashingTF, lr))
```

2）在训练集上拟合这个 Pipeline

```
val model = pipeline.fit(training)
```

3）在测试集上做预测。

```
model.transform(test).select("label", "prediction")
```

通过这种方式，既可保证 Stage 的有序执行，也可保证在训练集和测试集上所做逻辑操作的一致性，这里只是举了一个简单例子，下一章将详细介绍有关 Pipeline 的内容，同时在第 7 章后还有详细的使用实例。

2.9 模型选择或调优

在 ML 中一个重要的任务就是模型选择，或者使用给定的数据为给定的任务寻找最适合的模型或参数。这个过程又称调优。调优可以是对单个阶段进行调优，也可以一次性对

整个 Pipeline 进行调优。

MLlib 支持使用类似 CrossValidator 和 TrainValidationSplit 这样的工具进行模型选择。这些工具需要以下组件：

- Estimator：用户调优的算法或 Pipeline。
- ParamMap 集合：提供参数选择，有时又称用户查找的参数网格（parameter grid）。参数网格可以使用 ParamGridBuilder 来构建。
- Evaluator：衡量模型在测试数据上的拟合程度。

模型选择工具工作原理如下：

1）将输入数据划分为训练数据和测试数据。

2）对于每个（训练，测试）对，遍历一组 ParamMaps。用每一个 ParamMap 参数来拟合估计器，得到训练后的模型，再使用评估器来评估模型表现。

3）选择性能表现最优模型对应参数表。

2.9.1　交叉验证

交叉验证（CrossValidator）会将数据集切分成 K 折数据集合，分别用于训练和测试。例如，$K = 3$ 时，CrossValidator 会生成 3 个（训练数据，测试数据）对，每一个数据对的训练数据占 2/3，测试数据占 1/3。为了评估一个 ParamMap，CrossValidator 会计算这 3 个不同的（训练，测试）数据集对在 Estimator 拟合出的模型上的平均评估指标。

在找出最好的 ParamMap 后，CrossValidator 会利用此 ParamMap 在整个训练集上训练（fit）出一个泛化能力强、误差相对小的最佳模型，整个过程处于流程化管理之中，其工作流程如图 2-19 所示。

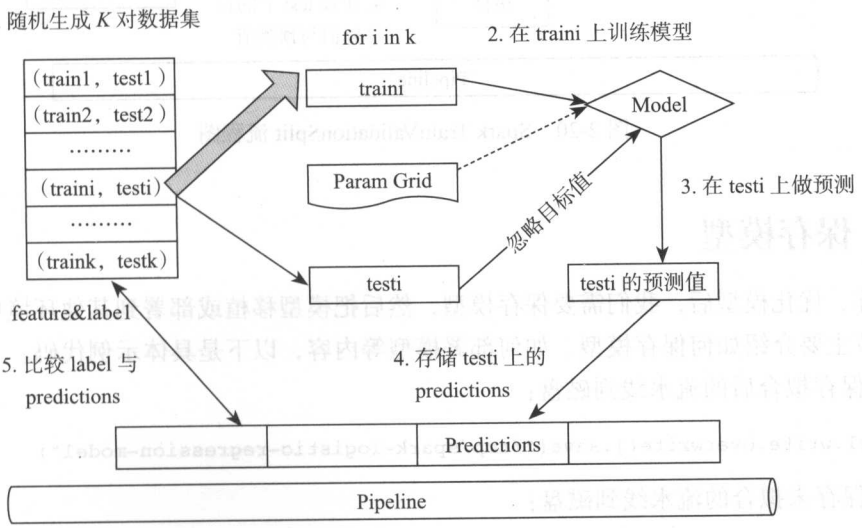


图 2-19　Spark CrossValidator 流程图

虽然利用 CrossValidator 来训练模型可以提升泛化能力，但其代价也比较高，如选择 $K = 3$、regParam = (0.1,0.01)、numIters = (10,20) 这样就需要对模型训练 $3 \times 2 \times 2 = 12$ 次。然而，对比启发式的手动调优，这是选择参数的行之有效的方法。

2.9.2 训练 – 验证切分

交叉验证的代价比较高昂，为此 Spark 也为超参数调优提供了训练 – 验证切分（TrainValidationSplit）。TrainValidationSplit 创建单一的（训练，测试）数据集对。它使用 trainRatio 参数将数据集切分成两部分。例如，当设置 trainRatio = 0.8 时，TrainValidationSplit 会对数据进行如下切分：80% 作为数据集，20% 作为验证集，以此来生成训练、测试集对，并最终使用最好的 ParamMap 和完整的数据集来拟合评估器。

相对于 CrossValidator 对每一个参数进行 K 次评估，TrainValidationSplit 只对每个参数组合评估 1 次（见图 2-20 所示）。因此它的评估代价没有那么高，但是当训练数据集不够大的时候其结果相对不够可信。

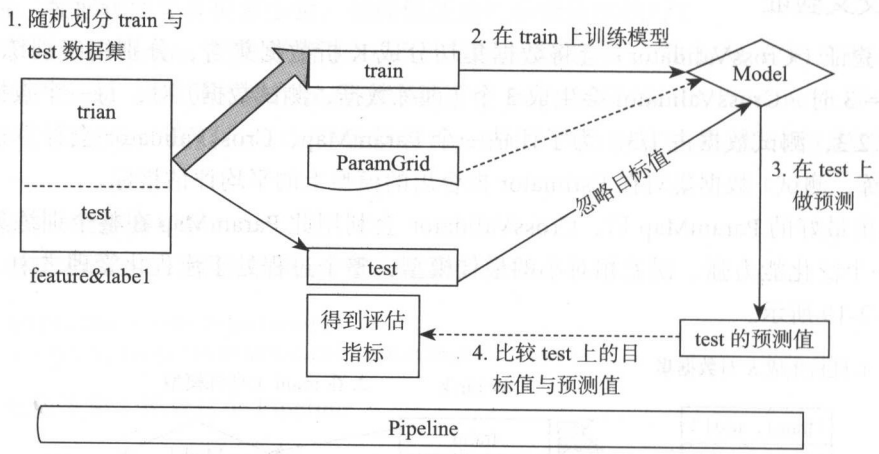


图 2-20 Spark TrainValidationSplit 流程图

2.10 保存模型

训练、优化模型后，我们需要保存模型，然后把模型移植或部署到其他环境中。本节主要介绍如何保存模型、如何部署模型等内容，以下是具体示例代码。

1）保存拟合后的流水线到磁盘：

```
model.write.overwrite().save("/tmp/spark-logistic-regression-model")
```

2）保存未拟合的流水线到磁盘：

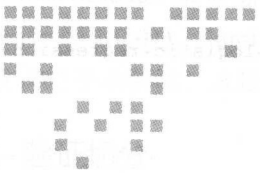
```
pipeline.write.overwrite().save("/tmp/spark-logistic-regression-model")
```

3）把拟合后的流水线部署到其他环境中：

```
val sameModel = PipelineModel.load("/tmp/spark-logistic-regression-model")
```

2.11　小结

这一章主要介绍了如何构建 Spark 学习系统以及构建的一般步骤等。实际上，构建 Spark 学习系统与我们构建其他平台的学习系统基本相同，一般都包括数据加载、数据探索、数据预测、构建模型、训练模型、评估模型、优化模型等步骤，但这里我们特别增加了利用 Pipeline 组装各个任务（stage）的内容，这也是 Spark ML 中基于 DataFrame 数据集的重要内容，下一章我们将详细介绍有关 Pipeline 的内容。



Chapter 3 第 3 章

ML Pipeline 原理与实战

Spark MLlib 是 Spark 的重要组成部分，也是最早推出的库之一，其基于 RDD 的 API，算法比较丰富，比较稳定，也比较好用。但是如果目标数据集结构复杂需要多次处理，或者是对新数据需要结合多个已经训练好的单个模型进行综合计算时，使用 MLlib 将会让程序结构复杂，甚至难于理解和实现。为改变这一局限性，从 Spark 1.2 版本之后引入了 ML Pipeline，经过多个版本的发展，Spark ML 克服了 MLlib 在处理复杂机器学习问题的一些不足（如工作比较复杂，流程不清晰等），向用户提供基于 DataFrame 之上的更加高层次的 API 库，以更加方便的构建复杂的机器学习工作流式应用，使整个机器学习过程变得更加易用、简洁、规范和高效。Spark 的 Pipeline 与 Scikit 中 Pipeline 的功能相近、理念相同。本章主要介绍 Spark ML 中 Pipeline 的有关内容。

本章主要介绍 ML Pipeline 的相关内容，包括：

- ❑ Pipeline 简介
- ❑ DataFrame
- ❑ 构成 Pipeline 的一些组件
- ❑ 介绍 pipeline 的一般原理
- ❑ 使用 pipeline 的几个实例

3.1 Pipeline 简介

ML 提倡使用 Pipeline，一般翻译为流水线，以便将多种算法更容易地组合成单个流水线或工作流程。一个 Pipeline 在结构上会包含一个或多个 Stage，每一个 Stage 都会完

成一个任务，如数据处理、数据转化、模型训练、参数设置或数据预测等，其中两个主要的 Stage 为 Transformer 和 Estimator。Transformer 主要是用来操作一个 DataFrame 数据并生成另外一个 DataFrame 数据，比如决策树模型、一个特征提取工具，都可以抽象为一个 Transformer。Estimator 则主要是用来做模型拟合，用来生成一个 Transformer。这些 Stage 有序组成一个 Pipeline。与 Pieline 相关的概念有：DataFrame、Transformer、Estimator、Parameter 等。

3.2　DataFrame

说到 DataFrame，使用过 Pandas、R 的朋友应该比较熟悉，它是一种类似于关系型数据库的表，使用和维护都非常方便。Spark 的 DataFrame 是基于早期版本中的 SchemaRDD，所以使用的是分布式大数据处理的场景。Spark DataFrame 以 RDD 为基础，但是带有 Schema 信息，可以从常规隐式地或明确地创建 RDD，它类似于传统数据库中的二维表格。它被 ML Pipeline 用来存储源数据。DataFrame 可以保存各种类型的数据，如我们可以把特征向量存储在 DataFrame 的一列中，这样用起来非常方便。

以下通过一个实例来说明 DataFrame 的创建、操作等内容：

```
// 以一个简单客户文件 customer.txt 为例，含 name,age,gender 三列
$cat customer.txt
name,age,gender
吴凡,25,F
张海燕,28,F
张宏,30,M
刘婷,29,F
高峰,31,M
// 创建 DataFrame
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.DataFrame

val spark = SparkSession.builder()
.appName("Spark SQL basic example")
.config("spark.some.config.option", "some-value")
.getOrCreate()
// 创建 DataFrame
val df1 = spark.read.option("header", true).format("csv").load("file:///home/
          hadoop/data/customer.csv")

// 转换字符类型
val df2 = df1.select(
    df1("name").cast("String"),
    df1("age").cast("Double"),
    df1("gender").cast("String"))
// 显示 df2 的 Schema
```

```
df2.printSchema()
root
  |-- name: string (nullable = true)
  |-- age: double (nullable = true)
  |-- gender: string (nullable = true)
// 创建视图，后续可以把视图作为表样使用
df2.createOrReplaceTempView("customer")

//查询
val cust1 = spark.sql("SELECT * FROM customer WHERE age BETWEEN 30 AND 35")
cust1.limit(5).show
+----+----+------+
|name| age|gender|
+----+----+------+
|  张宏|30.0|     M|
|  高峰|31.0|     M|
| 赵建军|32.0|     M|
|  李俊|32.0|     M|
| 孙达荣|33.0|     M|
+----+----+------+
val cust2 = spark.sql("SELECT * FROM customer WHERE gender like 'M'")
cust2.limit(5).show
+----+----+------+
|name| age|gender|
+----+----+------+
|  张宏|30.0|     M|
|  高峰|31.0|     M|
|  王华|40.0|     M|
| 赵建军|32.0|     M|
|  李俊|32.0|     M|
+----+----+------+
```

3.3 Pipeline 组件

前面提到 Pipeline 组件主要包括 Transformer 和 Estimator，下面来详细介绍。

1. Transformer

Transformer 一般翻译成转换器，是一个 Pipeline Stage，转换器包含特征变化和学习模型。从技术上来说，转化器通过方法 transform()，在原始数据上增加一列或者多列来将一个 DataFrame 转为另一个 DataFrame。如：

1）一个特征转换器输入一个 DataFrame，读取一个文本列，将其映射为新的特征向量列。输出一个新的带有特征向量列的 DataFrame。

2）一个学习模型转换器输入一个 DataFrame，读取包括特征向量的列，预测每一个特征向量的标签。输出一个新的带有预测标签列的 DataFrame。

2. Estimator

Estimator 可以被翻译成评估器或适配器，在 Pipeline 里通常是被用来操作 DataFrame 数据并生产一个 Transformer，一个分类算法就是一个 Estimator，因为它可以通过训练特征数据而得到一个分类模型。估计器用来拟合或者训练数据的学习算法或者任何算法。从技术上来说，估计器通过调用 fit() 方法，接受一个 DataFrame 产生一个模型，这个模型就是一个 Transformer。比如，逻辑回归就是一个估计器，通过调用 fit() 来产生一个逻辑回归模型。

3.4　Pipeline 原理

要构建一个 Pipeline，首先需要定义 Pipeline 中的各个 Stage，如指标提取和转换模型训练等。有了这些处理特定问题的 Transformer 和 Estimator，我们就可以按照具体的处理逻辑来有序地组织 Stage 并创建一个 Pipeline。

流水线由一系列有顺序的阶段指定，每个状态的运行是有顺序的，输入的 DataFrame 通过每个阶段进行改变。在转换器阶段，transform() 方法被调用于 DataFrame 上。对于估计器阶段，fit() 方法被调用来产生一个转换器，然后该转换器的 transform() 方法被调用在 DataFrame 上。图 3-1 简单说明了文档处理工作流的运行过程。

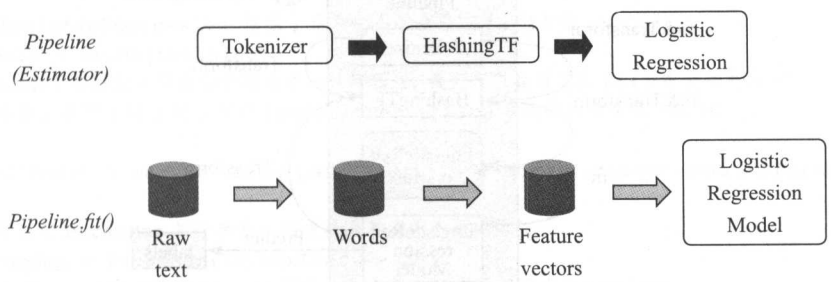


图 3-1　Pipeline 在训练数据上的流程

在图 3-1 中，第一行代表流水线处理的三个阶段。第一、二个阶段是转换器，第三个逻辑回归是估计器。底下一行代表流水线中的数据流，圆筒指 DataFrame。流水线的 fit() 方法被调用于原始的 DataFrame 中，里面包含原始的文档和标签。分词器的 transform() 方法将原始文档分为词语，添加新的词语列到 DataFrame 中。哈希处理的 transform() 方法将词语列转换为特征向量，添加新的向量列到 DataFrame 中。然后，因为逻辑回归是估计器，流水线先调用逻辑回归的 fit() 方法来产生逻辑回归模型。如果流水线还有其他更多阶段，在将 DataFrame 传入下一个阶段之前，流水线会先调用逻辑回归模型的 transform() 方法。

整个流水线是一个估计器。所以当流水线的 fit() 方法运行后，会产生一个流水线模型，流水线模型是转换器。流水线模型会在测试时被调用，图 3-2 说明了它的用法。

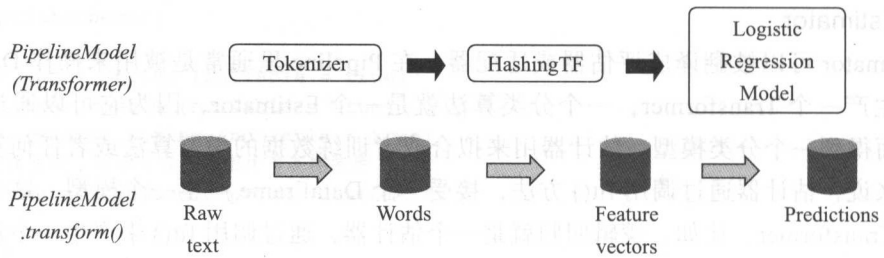


图 3-2 Pipeline 在测试数据上的流程

在图 3-2 中，流水线模型和原始流水线有同样数目的阶段，然而原始流水线中的估计器此时变为了转换器。当流水线模型的 transform() 方法被调用于测试数据集时，数据依次经过流水线的各个阶段。每个阶段的 transform() 方法更新数据集，并将之传到下个阶段。

流水线和流水线模型有助于确认训练数据和测试数据经过同样的特征处理流程。以上两图如果合并为一图，可用图 3-3 来表达。

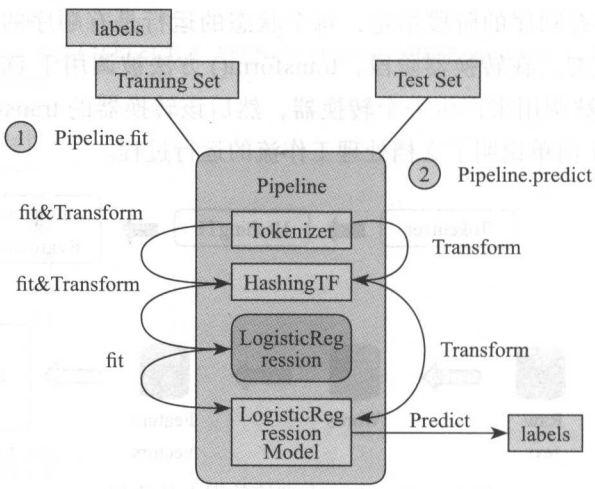


图 3-3 Spark pipeline 流程图

其中 Pipeline 及 LogisticRegression 为 Estimator，Tokenizer、HashingTF、LogisticRegression Model 为 Transformer。

3.5 Pipeline 实例

3.5.1 使用 Estimator、Transformer 和 Param 的实例

在机器学习整个过程中，特征转换、特征选择、派生特征等工作，一般需要占据大部分时间，现在 ML 提供了很多 Transformer，如 OneHotEncoder、StringIndexer、PCA、

Bucketizer、Word2vec 等，利用这些函数可极大提高工作效率。

以下通过实例说明如何使用 ML 库中的 Estimaor、Transformer 和 Param 等。

```
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.linalg.{Vector, Vectors}
import org.apache.spark.ml.param.ParamMap
import org.apache.spark.sql.Row

// 从（标识、特征）元组开始训练数据
val training = spark.createDataFrame(Seq(
    (1.0, Vectors.dense(0.0, 1.1, 0.1)),
    (0.0, Vectors.dense(2.0, 1.0, -1.0)),
    (0.0, Vectors.dense(2.0, 1.3, 1.0)),
    (1.0, Vectors.dense(0.0, 1.2, -0.5))
)).toDF("label", "features")

// 创建一个 LogisticRegression 实例，该实例是一个估计器
val lr = new LogisticRegression()
// 打印参数、文档和任何默认值
println("LogisticRegression parameters:\n" + lr.explainParams() + "\n")

// 我们可以使用 setter 方法设置参数
lr.setMaxIter(10)
    .setRegParam(0.01)

// 训练 LogisticRegression 模型，这里使用了存储在 lr 的参数
val model1 = lr.fit(training)
// 因为 model 是模型（即由估计器生成的转换器），我们可以查看它在 fit() 中使用的参数。
// 打印参数（名称：值）对，其中 LogisticRegression 实例的名称是唯一的 ID
//
println("Model 1 was fit using parameters: " + model1.parent.extractParamMap)

// 我们可以用 ParamMap 指令参数，它支持几种指定参数的方法
val paramMap = ParamMap(lr.maxIter -> 20)
    .put(lr.maxIter, 30)  // 指定 1 个参数，这会覆盖原来的 maxIter
    .put(lr.regParam -> 0.1, lr.threshold -> 0.55)  // 指定多个参数

// 也可以组合 paramMap
val paramMap2 = ParamMap(lr.probabilityCol -> "myProbability")  // 修改输出列名
val paramMapCombined = paramMap ++ paramMap2

// 现在使用 paramMapCombined 参数学习一个新的模型
// paramMapCombined 覆盖之前通过 lr.set* 方法设置的所有参数
val model2 = lr.fit(training, paramMapCombined)
println("Model 2 was fit using parameters: " + model2.parent.extractParamMap)

// 准备测试数据
val test = spark.createDataFrame(Seq(
    (1.0, Vectors.dense(-1.0, 1.5, 1.3)),
    (0.0, Vectors.dense(3.0, 2.0, -0.1)),
```

```
    (1.0, Vectors.dense(0.0, 2.2, -1.5))
)).toDF("label", "features")

// 使用 Transformer.transform() 方法对测试数据进行预测。
// LogisticRegression.transform 将仅使用 "特征" 列。
// 请注意 ,model2.transform() 输出一个 "myProbability" 列 , 而不是通常我们先前通过
// lr.probabilityCol 参数重命名的 'Probability 列。
model2.transform(test)
    .select("features", "label", "myProbability", "prediction")
    .collect()
    .foreach { case Row(features: Vector, label: Double, prob: Vector,
prediction: Double) =>
    println(s"($features, $label) -> prob=$prob, prediction=$prediction")
    }
```

3.5.2 ML 使用 Pipeline 的实例

要构建一个 Pipeline，首先我们需要定义 Pipeline 中的各个 Pipeline Stage，如分词、计算 TF-IDF 及训练逻辑回归模型等。这些 Transformer 和 Estimator 创建后，我们就可以按照处理流程组成 PipelineStages，并创建一个 Pipeline，如 val pipeline = new Pipeline().setStages(Array(stage1,stage2,…))。然后，把训练数据集作为参数并调用 Pipeline 实例的 fit() 方法，之后，将以流程的方式来处理原训练数据。以下是一个具体使用 Pipeline 的实例。

```
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.sql.Row

// 准备训练文档 (id, 内容 , 标签 )
val training = spark.createDataFrame(Seq(
    (0L, "a b c d e spark", 1.0),
    (1L, "b d", 0.0),
    (2L, "spark f g h", 1.0),
    (3L, "hadoop mapreduce", 0.0)
)).toDF("id", "text", "label")

// 配置 ML Pipeline, 由三个 stage 组成 ,tokenizer、hashingTF 和 lr
val tokenizer = new Tokenizer()
    .setInputCol("text")
    .setOutputCol("words")
val hashingTF = new HashingTF()
    .setNumFeatures(1000)
    .setInputCol(tokenizer.getOutputCol)
    .setOutputCol("features")
val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.001)
```

```
    val pipeline = new Pipeline()
        .setStages(Array(tokenizer, hashingTF, lr))

    //在训练数据集上使用 Pipeline
    val model = pipeline.fit(training)

    // Now we can optionally save the fitted pipeline to disk
    //现在可以保存安装好的流水线到磁盘上
    model.write.overwrite().save("/tmp/spark-logistic-regression-model")

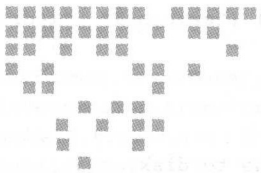
    //现在可以保存未安装好的 Pipeline 保存到磁盘上
    pipeline.write.overwrite().save("/tmp/unfit-lr-model")

    // 装载模型
    val sameModel = PipelineModel.load("/tmp/spark-logistic-regression-model")

    //准备测试文档，不包含标签 (id, text) .
    val test = spark.createDataFrame(Seq(
        (4L, "spark i j k"),
        (5L, "l m n"),
        (6L, "spark hadoop spark"),
        (7L, "apache hadoop")
    )).toDF("id", "text")
    //在测试文档上做出预测
    model.transform(test)
        .select("id", "text", "probability", "prediction")
        .collect()
        .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double)
=>
            println(s"($id, $text) --> prob=$prob, prediction=$prediction")
        }
```

3.6　小结

本章主要介绍了流水线（pipeline）的基本概念，以流水线的两个组件（Transformer 和 Estimator），它们是构成 Pipeline 的 Stage，把这些 Stage 按照一定次序组装到 Pipeline 上，就构成一个流水线，这些 Stage 包括特征转换、特征选择、模型训练等任务，然后通过几个实例具体说明 Pipeline 的创建及使用。流水线是一项重要内容，下一章将主要介绍构成 Pipeline 的一些 Stage。熟练使用这些 Stage 有助于提升我们的开发效率。



Chapter 4 第 4 章

特征提取、转换和选择

在实际机器学习项目中，我们获取的数据往往是不规范、不一致的，有很多缺失数据，甚至不少错误数据，这些数据有时又称为脏数据或噪声，在模型训练前，务必对这些脏数据进行处理，否则，再好的模型，也只能脏数据进，脏数据出。

这章我们主要介绍对数据处理涉及的一些操作，主要包括：

- ❑ 特征提取
- ❑ 特征转换
- ❑ 特征选择

4.1 特征提取

特征提取一般指从原始数据中抽取特征。

4.1.1 词频—逆向文件频率（TF-IDF）

词频—逆向文件频率（TF-IDF）是一种在文本挖掘中广泛使用的特征向量化方法，它可以体现一个文档中词语在语料库中的重要程度。

词语由 t 表示，文档由 d 表示，语料库由 D 表示。词频 TF(t,d) 是词语 t 在文档 d 中出现的次数。文件频率 DF(t,D) 是包含词语的文档的个数。如果我们只使用词频来衡量重要性，很容易过度强调在文档中经常出现而并没有包含太多与文档有关的信息的词语，比如“a”，“the”以及“of”。如果一个词语经常出现在语料库中，则意味着它并没有携带特定的文档的特殊信息。逆向文档频率数值化衡量词语提供多少信息：

$$IDF(t,D)=\log\frac{|D|+1}{DF(t,D)+1}$$

其中，|D| 是语料库中的文档总数。由于采用了对数，如果一个词出现在所有的文件，其 IDF 值变为 0。

$$TFIDF(t,d,D)=TF(t,d)\cdot IDF(t,D)$$

在下面的代码段中，我们以一组句子开始。首先使用分解器 Tokenizer 把句子划分为单个词语。对每一个句子（词袋），我们使用 HashingTF 将句子转换为特征向量，最后使用 IDF 重新调整特征向量。这种转换通常可以提高使用文本特征的性能。

```
import org.apache.spark.ml.feature.{HashingTF, IDF, Tokenizer}

val sentenceData = spark.createDataFrame(Seq(
  (0, "Hi I heard about Spark"),
  (0, "I wish Java could use case classes"),
  (1, "Logistic regression models are neat")
)).toDF("label", "sentence")

val tokenizer = new Tokenizer().setInputCol("sentence").setOutputCol("words")
val wordsData = tokenizer.transform(sentenceData)
val hashingTF = new
HashingTF().setInputCol("words").setOutputCol("rawFeatures").setNumFeatures(20)
val featurizedData = hashingTF.transform(wordsData)
// CountVectorizer 也可获取词频向量

val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")
val idfModel = idf.fit(featurizedData)
val rescaledData = idfModel.transform(featurizedData)
rescaledData.select("features", "label").take(3).foreach(println)
```

4.1.2　Word2Vec

Word2vec 是一个 Estimator，它采用一系列代表文档的词语来训练 word2vecmodel。该模型将每个词语映射到一个固定大小的向量。word2vecmodel 使用文档中每个词语的平均数来将文档转换为向量，然后这个向量可以作为预测的特征，来计算文档相似度计算等。

在下面的代码段中，我们用一组文档开始，其中每一个文档代表一个词语序列。对于每一个文档，我们将其转换为一个特征向量。此特征向量可以被传递到一个学习算法中。

```
import org.apache.spark.ml.feature.Word2Vec

// 输入数据，每行为一个词袋，可来自语句或文档
val documentDF = spark.createDataFrame(Seq(
  "Hi I heard about Spark".split(" "),
  "I wish Java could use case classes".split(" "),
  "Logistic regression models are neat".split(" ")
```

```
).map(Tuple1.apply)).toDF("text")

// 训练从词到向量的映射
val word2Vec = new Word2Vec()
  .setInputCol("text")
  .setOutputCol("result")
  .setVectorSize(3)
  .setMinCount(0)
val model = word2Vec.fit(documentDF)
val result = model.transform(documentDF)
result.select("result").take(3).foreach(println)
```

4.1.3 计数向量器

计数向量器（Countvectorizer）和计数向量器模型（Countvectorizermodel）旨在通过计数来将一个文档转换为向量。当不存在先验字典时，Countvectorizer 可作为 Estimator 来提取词汇，并生成一个 Countvectorizermodel。该模型通过词汇生成文档的稀疏表示，然后可以将其传递给其他算法，如 LDA。

在 fitting 过程中，countvectorizer 将根据语料库中的词频排序选出前 vocabsize 个词。一个可选的参数 minDF 也影响 fitting 过程，它指定词汇表中的词语在文档中最少出现的次数。另一个可选的二值参数控制输出向量，如果设置为真，那么所有非零的计数为 1，这对于二值型离散概率模型非常有用。

下面用实例来说明计数向量器的使用。假设有以下列 id 和 texts 构成的 DataFrame：

```
id | texts
----|----------
 0  | Array("a", "b", "c")
 1  | Array("a", "b", "b", "c", "a")
```

每行 text 都是 Array [String] 类型的文档。调用 fit，CountVectorizer 产生 CountVectorizerModel。转换后的输出列“向量”包含：

调用的 CountVectorizer 产生词汇（a,b,c）的 CountVectorizerModel，转换后的输出向量如下：

```
id | texts                           | vector
---|---------------------------------|---------------
 0  | Array("a", "b", "c")            | (3,[0,1,2],[1.0,1.0,1.0])
 1  | Array("a", "b", "b", "c", "a") | (3,[0,1,2],[2.0,2.0,1.0])
```

每个向量代表文档的词汇表中每个词语出现的次数。

```
importorg.apache.spark.ml.feature.{CountVectorizer,CountVectorizerModel}

val df = spark.createDataFrame(Seq(
(0,Array("a","b","c")),
(1,Array("a","b","b","c","a"))
)).toDF("id","words")
```

```
// 从语料库中拟合 CountVectorizerModel
val cvModel:CountVectorizerModel=newCountVectorizer()
.setInputCol("words")
.setOutputCol("features")
.setVocabSize(3)
.setMinDF(2)
.fit(df)

// 也可以用先验词汇表定义 CountVectorizerModel
val cvm =newCountVectorizerModel(Array("a","b","c"))
.setInputCol("words")
.setOutputCol("features")

cvModel.transform(df).show(false)
+---+---------------+-------------------------+
|id |words          |features                 |
+---+---------------+-------------------------+
|0  |[a, b, c]      |(3,[0,1,2],[1.0,1.0,1.0])|
|1  |[a, b, b, c, a]|(3,[0,1,2],[2.0,2.0,1.0])
|+---+---------------+-------------------------+
```

4.2　特征转换

在机器学习中，数据处理是一件比较繁琐的事情，需要对原有特征做多种处理，如类型转换、标准化特征、新增衍生特征等，需要耗费大量的时间和精力编写处理程序。不过，自从 Spark 推出 ML 后，情况大有改观。Spark ML 包中提供了很多现成的转换器，例如，StringIndexer、IndexToString、OneHotEncoder、VectorIndexer，提供了十分方便的特征转换功能，这些转换器类都位于 org.apache.spark.ml.feature 包下。

4.2.1　分词器

分词器（Tokenization）将文本划分为独立个体（通常为单词）。下面的例子展示了如何把句子划分为单词。RegexTokenizer 基于正则表达式提供更多的划分选项。默认情况下，参数“pattern”（regex, default: “\\s+”）为划分文本的分隔符。或者，用户可以指定参数“gaps”来指明正则“patten”表示的是“tokens”而不是分隔符，这样来为分词结果找到所有可能匹配的情况。

```
importorg.apache.spark.ml.feature.{RegexTokenizer,Tokenizer}
importorg.apache.spark.sql.functions._

val sentenceDataFrame = spark.createDataFrame(Seq(
(0,"Hi I heard about Spark"),
(1,"I wish Java could use case classes"),
```

```
(2,"Logistic,regression,models,are,neat")
)).toDF("id","sentence")

val tokenizer =newTokenizer().setInputCol("sentence").setOutputCol("words")
val regexTokenizer =newRegexTokenizer()
.setInputCol("sentence")
.setOutputCol("words")
.setPattern("\\W") // 或者使用 .setPattern("\\w+").setGaps(false)

val countTokens = udf {(words:Seq[String])=> words.length }

val tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence","words")
.withColumn("tokens", countTokens(col("words"))).show(false)
+---------------------------------+-----------------------------------------+------+
|sentence                         |words                                    |tokens|
+---------------------------------+-----------------------------------------+------+
|Hi I heard about Spark           |[hi, i, heard, about, spark]             |5     |
|I wish Java could use case classes|[i, wish, java, could, use, case, classes]|7     |
|Logistic,regression,models,are,neat|[logistic,regression,models,are,neat]   |1     |
+---------------------------------+-----------------------------------------+------+
val regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence","words")
.withColumn("tokens", countTokens(col("words"))).show(false)
+---------------------------------+-----------------------------------------+------+
|sentence                         |words                                    |tokens|
+---------------------------------+-----------------------------------------+------+
|Hi I heard about Spark           |[hi, i, heard, about, spark]             |5     |
|I wish Java could use case classes|[i, wish, java, could, use, case, classes]|7     |
|Logistic,regression,models,are,neat|[logistic, regression, models, are, neat]|5     |
+---------------------------------+-----------------------------------------+------+
```

4.2.2 移除停用词

停用词是在文档中频繁出现，但未承载太多意义的词语，它们不应该被包含在算法输入中，所以会用到移除停用词（StopWordsRemover）。StopWordsRemover 的输入为一系列字符串（如分词器输出），输出中删除了所有停用词。停用词表由 stopWords 参数提供。一些语言的默认停用词表可以通过 StopWordsRemover.loadDefaultStopWords(language) 调用。布尔参数 caseSensitive 指明是否区分大小写（默认为否）。下面是一些具体示例。

假设我们有如下 DataFrame，有 id 和 raw 两列：

```
 id  | raw
-----|----------
  0  | [I, saw, the, red, baloon]
  1  | [Mary, had, a, little, lamb]
```

通过对 raw 列调用 StopWordsRemover，我们可以得到筛选出的结果列：

```
 id  | raw                                | filtered
-----|------------------------------------|--------------------
  0  | [I, saw, the, red, baloon]  |  [saw, red, baloon]
  1  | [Mary, had, a, little, lamb]|[Mary, little, lamb]
```

其中，“I”“the”“had”以及“a”被移除。

实现以上功能的详细代码如下：

```
import org.apache.spark.ml.feature.StopWordsRemover

val remover = new StopWordsRemover()
  .setInputCol("raw")
  .setOutputCol("filtered")

val dataSet = spark.createDataFrame(Seq(
  (0, Seq("I", "saw", "the", "red", "balloon")),
  (1, Seq("Mary", "had", "a", "little", "lamb"))
)).toDF("id", "raw")
remover.transform(dataSet).show(false)
```

4.2.3　n-gram

一个 n-gram 是一个长度为整数 n 的字序列。NGram 可以用来将输入转换为 n-gram。NGram 的输入为一系列字符串（如 Tokenizer 的输出）。参数 n 决定每个 n-gram 包含的对象个数。结果包含一系列 n-gram，其中每个 n-gram 代表一个空格分割的 n 个连续字符。如果输入少于 n 个字符串，将没有输出结果。

```
import org.apache.spark.ml.feature.NGram

val wordDataFrame = spark.createDataFrame(Seq(
  (0, Array("Hi", "I", "heard", "about", "Spark")),
  (1, Array("I", "wish", "Java", "could", "use", "case", "classes")),
  (2, Array("Logistic", "regression", "models", "are", "neat"))
)).toDF("id", "words")

val ngram = new NGram().setN(2).setInputCol("words").setOutputCol("ngrams")

val ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.select("ngrams").show(false)
+------------------------------------------------------------------+
|ngrams                                                            |
+------------------------------------------------------------------+
|[Hi I, I heard, heard about, about Spark]                         |
|[I wish, wish Java, Java could, could use, use case, case classes]|
|[Logistic regression, regression models, models are, are neat]    |
+------------------------------------------------------------------+
```

4.2.4 二值化

二值化，即通过设置阈值，将连续型的特征转化为两个值。大于阈值为 1，否则为 0。注：以下规范化操作一般是针对一个特征向量（dataFrame 中的一个 colum）来操作的。

```
import org.apache.spark.ml.feature.Binarizer

val data = Array((0, 0.1), (1, 0.8), (2, 0.2))
val dataFrame = spark.createDataFrame(data).toDF("id", "feature")
+---+-------+
| id|feature|
+---+-------+
|  0|    0.1|
|  1|    0.8|
|  2|    0.2|
+---+-------+
val binarizer: Binarizer = new Binarizer()
  .setInputCol("feature")
  .setOutputCol("binarized_feature")
  .setThreshold(0.5)

val binarizedDataFrame = binarizer.transform(dataFrame)

println(s"Binarizer output with Threshold = ${binarizer.getThreshold}")
binarizedDataFrame.show()
+---+-------+-----------------+
| id|feature|binarized_feature|
+---+-------+-----------------+
|  0|    0.1|              0.0|
|  1|    0.8|              1.0|
|  2|    0.2|              0.0|
+---+-------+-----------------+
```

4.2.5 主成分分析

主成分分析（PCA）是一种统计学方法，其本质是在线性空间中进行一个基变换，使得变换后的数据投影在一组新的“坐标轴”上的方差最大化，然后，依据变换后方差大小确定“坐标轴”的权重或重要性，权重高的被称为主成分（Principal Component），用变换后的一个较低维度的子空间代表原有数据的特征。主成分分析被广泛应用在各种统计学、机器学习问题中，是最常见的降维方法之一。

PCA 在 Spark2.0 用法比较简单，只需要设置：

```
.setInputCol("features")// 保证输入是特征值向量
.setOutputCol("pcaFeatures")// 输出
.setK(3)// 主成分个数
```

注意　PCA 前一定要对特征向量进行规范化（标准化）！！！

```
import org.apache.spark.ml.feature.PCA
import org.apache.spark.ml.feature.PCAModel          //不是 mllib
import org.apache.spark.ml.feature.StandardScaler
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.Row
import org.apache.spark.sql.SparkSession

val rawDataFrame=spark.read.format("libsvm").load("file:///home/hadoop/bigdata/
spark/data/mllib/sample_libsvm_data.txt")

val scaledDataFrame=new StandardScaler()
    .setInputCol("features")
    .setOutputCol("scaledFeatures")
    .setWithMean(false)          //对于稀疏数据（如本次使用的数据），不要使用平均值
    .setWithStd(true)
    .fit(rawDataFrame)
    .transform(rawDataFrame)
//PCA 模型
val pcaModel=new PCA().setInputCol("scaledFeatures")
                      .setOutputCol("pcaFeatures")
                      .setK(3)//
                      .fit(scaledDataFrame)
//进行 PCA 降维
pcaModel.transform(scaledDataFrame).select("label","pcaFeatures").show(10,false)
//没有标准化特征向量，直接进行 PCA 主成分：各主成分之间值变化太大，有数量级的差别。标准化特征向
//量后 PCA 主成分，各主成分之间值基本上在同一水平上，结果更合理
+-----+-----------------------------------------------------------+
|label|pcaFeatures                                                 |
+-----+-----------------------------------------------------------+
|0.0  |[-14.998868464839624,-10.137788261664621,-3.042873539670117]|
|1.0  |[2.1965800525589754,-4.139257418439533,-11.386135042845101] |
|1.0  |[1.0254645688925883,-0.8905813756164163,7.168759904518129]  |
|1.0  |[1.5069317554093433,-0.7289177578028571,5.23152743564543]   |
|1.0  |[1.6938250375084654,-0.4350617717494331,4.770263568537382]  |
|0.0  |[-15.870371979062549,-9.999445137658528,-6.521920373215663]
//如何选择 k 值?
val  pcaModel=new PCA().setInputCol("scaledFeatures")
    .setOutputCol("pcaFeatures")
    .setK(50)//
    .fit(scaledDataFrame)
var i=1
for( x<-pcaModel.explainedVariance.toArray){
    println(i+"\t"+x+"  ")
    i +=1
    }
//运行结果（前 10 行），随着 k 的增加，精度趋于平稳
```

```
1       0.25934799275530857
2       0.12355355301486977
3       0.07447670060988294
4       0.0554545717486928
5       0.04207050513264405
6       0.03715986573644129
7       0.031350566055423544
8       0.027797304129489515
9       0.023825873477496748
10      0.02268054946233242
```

4.2.6 多项式展开

多项式展开（Polynomial Expansion）即通过产生 n 维组合将原始特征扩展到多项式空间。下面的示例会介绍如何将你的特征集拓展到 3 维多项式空间。

```
import org.apache.spark.ml.feature.PolynomialExpansion
import org.apache.spark.ml.linalg.Vectors

val data = Array(
  Vectors.dense(2.0, 1.0),
  Vectors.dense(0.0, 0.0),
  Vectors.dense(3.0, -1.0)
)
val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

val polyExpansion = new PolynomialExpansion()
  .setInputCol("features")
  .setOutputCol("polyFeatures")
  .setDegree(3)

val polyDF = polyExpansion.transform(df)
polyDF.show(false)
+----------+-----------------------------------------+
|features  |polyFeatures                             |
+----------+-----------------------------------------+
|[2.0,1.0] |[2.0,4.0,8.0,1.0,2.0,4.0,1.0,2.0,1.0]    |
|[0.0,0.0] |[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]    |
|[3.0,-1.0]|[3.0,9.0,27.0,-1.0,-3.0,-9.0,1.0,3.0,-1.0]|
+----------+-----------------------------------------+
```

4.2.7 离散余弦变换

离散余弦变换（DCT）是与傅里叶变换相关的一种变换，它类似于离散傅立叶变换，但是只使用实数。离散余弦变换相当于一个长度大概是它两倍的离散傅里叶变换，这个离散傅里叶变换是对一个实偶函数进行的（因为一个实偶函数的傅里叶变换仍然是一个实偶函数）。离散余弦变换，经常用于信号处理和图像处理中，对信号和图像（包括静止图像和运

动图像）进行有损数据压缩。

```
import org.apache.spark.ml.feature.DCT
import org.apache.spark.ml.linalg.Vectors

val data = Seq(
  Vectors.dense(0.0, 1.0, -2.0, 3.0),
  Vectors.dense(-1.0, 2.0, 4.0, -7.0),
  Vectors.dense(14.0, -2.0, -5.0, 1.0))

val df = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

val dct = new DCT()
  .setInputCol("features")
  .setOutputCol("featuresDCT")
  .setInverse(false)

val dctDf = dct.transform(df)
dctDf.select("featuresDCT").show(false)
+------------------------------------------------------------------+
|featuresDCT                                                       |
+------------------------------------------------------------------+
|[1.0,-1.1480502970952693,2.000000000000004,-2.7716385975338604]|
|[-1.0,3.378492794482933,-7.000000000000001,2.9301512653149677]  |
|[4.0,9.304453421915744,11.000000000000002,1.5579302036357163]   |
+------------------------------------------------------------------+
```

4.2.8　字符串—索引变换

字符串—索引变换（StringIndexer）是将字符串列编码为标签索引列。索引位于 [0, numLabels)，按标签频率排序，因此最常见的标签获取索引 0。如果输入列是数值型，我们将其转换为字符串并索引字符串值。当下游流水线组件（如 Estimator 或 Transformer）使用此字符串索引标签时，必须将组件的输入列设置为此字符串索引的列名称。通常可以使用 setInputCol 设置输入列。

示例数据为一个含有 id 和 category 两列的 DataFrame，如下：

```
id  | category
----|----------
 0  | a
 1  | b
 2  | c
 3  | a
 4  | a
 5  | c
```

category 是有 3 种取值的字符串列（a、b、c），使用 StringIndexer 进行转换后我们可以得到如下输出，其中 category 作为输入列，categoryIndex 作为输出列：

```
id  | category | categoryIndex
----|----------|---------------
 0  | a        | 0.0
 1  | b        | 2.0
 2  | c        | 1.0
 3  | a        | 0.0
 4  | a        | 0.0
 5  | c        | 1.0
```

a 获得索引 0，因为它是最频繁的，随后是具有索引 1 的 c 和具有索引 2 的 b。此外，有两个策略，如果在转换新数据（如测试数据集）时出现了在训练中未出现的标签，StringIndexer 将会报错（默认值）或者跳过未出现的标签实例（即 setHandleInvalid ("skip")）。

例如，测试数据集中比训练数据集多了一个 d 类：

```
id  | category
----|----------
 0  | a
 1  | b
 2  | c
 3  | d
```

如果你没有设置 StringIndexer 如何处理未看见的标签（默认值）或没有将其设置为“错误”，则会抛出异常。但是，如果你调用了 setHandleInvalid ("skip")，d 类将不出现，结果为以下数据集：

```
id  | category | categoryIndex
----|----------|---------------
 0  | a        | 0.0
 1  | b        | 2.0
 2  | c        | 1.0
```

以下是使用 StringIndexer 的一个示例：

```
import org.apache.spark.ml.feature.StringIndexer

val df = spark.createDataFrame(
  Seq((0, "a"), (1, "b"), (2, "c"), (3, "a"), (4, "a"), (5, "c"))
).toDF("id", "category")

val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")

val indexed = indexer.fit(df).transform(df)
indexed.show()
+---+--------+-------------+
| id|category|categoryIndex|
```

```
+---+--------+-------------+
|  0|       a|          0.0|
|  1|       b|          2.0|
|  2|       c|          1.0|
|  3|       a|          0.0|
|  4|       a|          0.0|
|  5|       c|          1.0|
+---+--------+-------------+
```

4.2.9　索引—字符串变换

与 StringIndexer 对应，索引—字符串变换（IndexToString）是将指标标签映射回原始字符串标签。一个常见的用例是使用 StringIndexer 从标签生成索引，使用这些索引训练模型，并从 IndexToString 的预测索引列中检索原始标签。但是，你可以自由提供你自己的标签。下面是一个基于 StringIndexer 的示例，假设我们有以下 DataFrame，其中包含的列为 id 和 categoryIndex：

```
id  | categoryIndex
----|---------------
 0  | 0.0
 1  | 2.0
 2  | 1.0
 3  | 0.0
 4  | 0.0
 5  | 1.0
```

应用 IndexToString，将 categoryIndex 作为输入列，将 originalCategory 作为输出列，我们可以检索我们的原始标签（它们将从列的元数据中推断）：

```
id  | categoryIndex | originalCategory
----|---------------|-----------------
 0  | 0.0           | a
 1  | 2.0           | b
 2  | 1.0           | c
 3  | 0.0           | a
 4  | 0.0           | a
 5  | 1.0           | c
```

下面是一个完整实例：

```
import org.apache.spark.ml.attribute.Attribute
import org.apache.spark.ml.feature.{IndexToString, StringIndexer}

val df = spark.createDataFrame(Seq(
  (0, "a"),
  (1, "b"),
  (2, "c"),
  (3, "a"),
```

```
  (4, "a"),
  (5, "c")
)).toDF("id", "category")

val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")
  .fit(df)
val indexed = indexer.transform(df)

println(s"Transformed string column '${indexer.getInputCol}' " +
    s"to indexed column '${indexer.getOutputCol}'")
indexed.show()

val inputColSchema = indexed.schema(indexer.getOutputCol)
println(s"StringIndexer will store labels in output column metadata: " +
    s"${Attribute.fromStructField(inputColSchema).toString}\n")

val converter = new IndexToString()
  .setInputCol("categoryIndex")
  .setOutputCol("originalCategory")

val converted = converter.transform(indexed)

println(s"Transformed indexed column '${converter.getInputCol}' back to original
string " +
    s"column '${converter.getOutputCol}' using labels in metadata")
converted.select("id", "categoryIndex", "originalCategory").show()
```

4.2.10 独热编码

独热编码（OneHotEncoder）将标签指标映射为二值向量，其中最多一个单值。这种编码可以使期望连续特征的算法使用分类特征，如逻辑回归等。

```
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}

val df = spark.createDataFrame(Seq(
  (0, "a"),
  (1, "b"),
  (2, "c"),
  (3, "a"),
  (4, "a"),
  (5, "c")
)).toDF("id", "category")

val indexer = new StringIndexer()
  .setInputCol("category")
  .setOutputCol("categoryIndex")
  .fit(df)
```

```
val indexed = indexer.transform(df)

val encoder = new OneHotEncoder()
  .setInputCol("categoryIndex")
  .setOutputCol("categoryVec")
  .setDropLast(false)

val encoded = encoder.transform(indexed)
encoded.show()
```

【说明】

1）OneHotEncoder 缺省状态下将删除最后一个分类或把最后一个分类作为 0。下面是一个具体示例：

```
import org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}

val fd = spark.createDataFrame( Seq((1.0, "a"), (1.5, "a"), (10.0, "b"), (3.2,
      "c"),(3.8,"c"))).toDF("x","c")
val ss =new StringIndexer().setInputCol("c").setOutputCol("c_idx")
val ff = ss.fit(fd).transform(fd)
ff.show()
```

显示结果如下：

```
+----+---+-----+
|   x|  c|c_idx|
+----+---+-----+
| 1.0|  a|  0.0|
| 1.5|  a|  0.0|
|10.0|  b|  2.0|
| 3.2|  c|  1.0|
| 3.8|  c|  1.0|
+----+---+-----+
```

最后一个分类为 b，通过 OneHotEncoder 变为向量后，已被删除。

```
val oe = new OneHotEncoder().setInputCol("c_idx").setOutputCol("c_idx_vec")
val fe = oe.transform(ff)
fe.show()
```

显示结果如下：

```
+----+---+-----+-------------+
|   x|  c|c_idx|    c_idx_vec|
+----+---+-----+-------------+
| 1.0|  a|  0.0|(2,[0],[1.0])|
| 1.5|  a|  0.0|(2,[0],[1.0])|
|10.0|  b|  2.0|    (2,[],[])|
| 3.2|  c|  1.0|(2,[1],[1.0])|
| 3.8|  c|  1.0|(2,[1],[1.0])|
+----+---+-----+-------------+
```

与其他特征组合为特征向量后，将置为 0，请看如下示例：

```
val assembler = new VectorAssembler().setInputCols(Array("x", "c_idx", "c_idx_
                vec")).setOutputCol("features")
val vecDF: DataFrame = assembler.transform(fe)
vecDF.show(false)
```

显示结果如下：

```
+----+---+-----+-------------+------------------+
|x   |c  |c_idx|c_idx_vec    |features          |
+----+---+-----+-------------+------------------+
|1.0 |a  |0.0  |(2,[0],[1.0])|[1.0,0.0,1.0,0.0] |
|1.5 |a  |0.0  |(2,[0],[1.0])|[1.5,0.0,1.0,0.0] |
|10.0|b  |2.0  |(2,[],[])    |[10.0,2.0,0.0,0.0]|
|3.2 |c  |1.0  |(2,[1],[1.0])|[3.2,1.0,0.0,1.0] |
|3.8 |c  |1.0  |(2,[1],[1.0])|[3.8,1.0,0.0,1.0] |
+----+---+-----+-------------+------------------+
```

如果想不删除最后一个分类，可添加 setDropLast（False）：

```
oe.setDropLast(false)
val fl = oe.transform(ff)
fl.show()
```

显示结果如下：

```
+----+---+-----+-------------+
|   x|  c|c_idx|    c_idx_vec|
+----+---+-----+-------------+
| 1.0|  a|  0.0|(3,[0],[1.0])|
| 1.5|  a|  0.0|(3,[0],[1.0])|
|10.0|  b|  2.0|(3,[2],[1.0])|
| 3.2|  c|  1.0|(3,[1],[1.0])|
| 3.8|  c|  1.0|(3,[1],[1.0])|
+----+---+-----+-------------+
```

与其他特征向量结合后，情况如下：

```
val vecDFl: DataFrame = assembler.transform(fl)
vecDFl.show(false)
```

显示结果如下：

```
+----+---+-----+-------------+----------------------+
|x   |c  |c_idx|c_idx_vec    |features              |
+----+---+-----+-------------+----------------------+
|1.0 |a  |0.0  |(3,[0],[1.0])|(5,[0,2],[1.0,1.0])   |
|1.5 |a  |0.0  |(3,[0],[1.0])|(5,[0,2],[1.5,1.0])   |
|10.0|b  |2.0  |(3,[2],[1.0])|[10.0,2.0,0.0,0.0,1.0]|
|3.2 |c  |1.0  |(3,[1],[1.0])|[3.2,1.0,0.0,1.0,0.0] |
|3.8 |c  |1.0  |(3,[1],[1.0])|[3.8,1.0,0.0,1.0,0.0] |
+----+---+-----+-------------+----------------------+
```

2）果分类中出现空字符，需要进行处理，如设置为 "None"，否则会报错。

4.2.11　向量—索引变换

向量—索引变换（VectorIndexer）能提高决策树或随机森林等 ML 方法的分类效果，是对数据集特征向量中的类别（离散值）特征（index categorical feature）进行编号。它能够自动判断哪些特征是离散值型的特征，并对它们进行编号。具体来说，它执行以下操作：

1）获取 Vector 类型的输入列和参数 maxCategories。

2）基于不同值的数量确定哪些特征应是分类的，其中最多 maxCategories 的特征被声明为分类特征。

3）为每个分类特征计算基于 0 的类别索引。

4）索引分类特征并将原始特征值转换为索引。

5）索引分类特征允许诸如决策树和树集合等算法适当地处理分类特征，从而提高性能。在下面的例子中，我们读取一个标记点的数据集，然后使用 VectorIndexer 来决定哪些特征应该被视为分类特征。我们将分类特征值转换为它们的索引。这个变换的数据然后可以被传递到诸如 DecisionTreeRegressor 的处理分类特征的算法。

```
import org.apache.spark.ml.feature.VectorIndexer

val data = spark.read.format("libsvm").load("file:///home/hadoop/bigdata/spark/
        data/mllib/sample_libsvm_data.txt")

val indexer = new VectorIndexer()
  .setInputCol("features")
  .setOutputCol("indexed")
  .setMaxCategories(10)

val indexerModel = indexer.fit(data)

val categoricalFeatures: Set[Int] = indexerModel.categoryMaps.keys.toSet
println(s"Chose ${categoricalFeatures.size} categorical features: " +
  categoricalFeatures.mkString(", "))

// 创建一个新列，"索引"列，将分类特征转换为索引
val indexedData = indexerModel.transform(data)
indexedData.show()
+-----+--------------------+--------------------+
|label|            features|             indexed|
+-----+--------------------+--------------------+
|  0.0|(692,[127,128,129...|(692,[127,128,129...|
|  1.0|(692,[158,159,160...|(692,[158,159,160...|
|  1.0|(692,[124,125,126...|(692,[124,125,126...|
|  1.0|(692,[152,153,154...|(692,[152,153,154...|
|  1.0|(692,[151,152,153...|(692,[151,152,153...|
|  0.0|(692,[129,130,131...|(692,[129,130,131...|
```

4.2.12 交互式

交互（Interaction）是一个变换器，它使用向量或双值列，并生成单个向量列，其中包含每个输入列的一个值的所有组合的乘积。例如，如果你有两个向量类型列，每个列有 3 个维度作为输入列，那么你将获得一个 9 维向量作为输出列。

假设我们有以下 DataFrame，包含 id1、vec1 和 vec2 三列：

```
id1|vec1          |vec2
---|--------------|--------------
1  |[1.0,2.0,3.0] |[8.0,4.0,5.0]
2  |[4.0,3.0,8.0] |[7.0,9.0,8.0]
3  |[6.0,1.0,9.0] |[2.0,3.0,6.0]
4  |[10.0,8.0,6.0]|[9.0,4.0,5.0]
5  |[9.0,2.0,7.0] |[10.0,7.0,3.0]
6  |[1.0,1.0,4.0] |[2.0,8.0,4.0]
```

应用与这些输入列的交互，然后将 interactionedCol 作为输出列，结果如下：

```
id1|vec1          |vec2          |interactedCol
---|--------------|--------------|-----------------------------------------------------
1  |[1.0,2.0,3.0] |[8.0,4.0,5.0] |[8.0,4.0,5.0,16.0,8.0,10.0,24.0,12.0,15.0]
2  |[4.0,3.0,8.0] |[7.0,9.0,8.0] |[56.0,72.0,64.0,42.0,54.0,48.0,112.0,144.0,128.0]
3  |[6.0,1.0,9.0] |[2.0,3.0,6.0] |[36.0,54.0,108.0,6.0,9.0,18.0,54.0,81.0,162.0]
4  |[10.0,8.0,6.0]|[9.0,4.0,5.0] |[360.0,160.0,200.0,288.0,128.0,160.0,216.0,96.0,120.0]
5  |[9.0,2.0,7.0] |[10.0,7.0,3.0]|[450.0,315.0,135.0,100.0,70.0,30.0,350.0,245.0,105.0]
6  |[1.0,1.0,4.0] |[2.0,8.0,4.0] |[12.0,48.0,24.0,12.0,48.0,24.0,48.0,192.0,96.0]
```

以下是实现以上转换的具体代码：

```
import org.apache.spark.ml.feature.Interaction
import org.apache.spark.ml.feature.VectorAssembler

val df = spark.createDataFrame(Seq(
  (1, 1, 2, 3, 8, 4, 5),
  (2, 4, 3, 8, 7, 9, 8),
  (3, 6, 1, 9, 2, 3, 6),
  (4, 10, 8, 6, 9, 4, 5),
  (5, 9, 2, 7, 10, 7, 3),
  (6, 1, 1, 4, 2, 8, 4)
)).toDF("id1", "id2", "id3", "id4", "id5", "id6", "id7")

val assembler1 = new VectorAssembler().
  setInputCols(Array("id2", "id3", "id4")).
  setOutputCol("vec1")

val assembled1 = assembler1.transform(df)

val assembler2 = new VectorAssembler().
  setInputCols(Array("id5", "id6", "id7")).
```

```
  setOutputCol("vec2")

val assembled2 = assembler2.transform(assembled1).select("id1", "vec1", "vec2")

val interaction = new Interaction()
  .setInputCols(Array("id1", "vec1", "vec2"))
  .setOutputCol("interactedCol")

val interacted = interaction.transform(assembled2)

interacted.show(truncate = false)
```

4.2.13　正则化

正则化（Normalizer）是一个变换器，用于转换向量（特征向量）的数据集，规范每个向量以具有单位范数。它采用参数 p 实现归一化，默认情况下 p = 2。这种归一化可以帮助标准化输入数据并改进学习算法的性能。

最常用的范数就是 p- 范数。若 $x=[x_1,x_2\cdots,x_n]^T$，那么

$$\|x\|_p=\left(|x_1|^p+|x_2|^p+\cdots+|x_n|^p\right)^{\frac{1}{p}}$$

当 p 取 1、2、∞的时候分别是以下几种最简单的情形：

- 1- 范数：‖ x ‖ 1= | x_1 | + | x_2 | +⋯+ | x_n |
- 2- 范数：‖ x ‖ 2=（| x_1 | 2+ | x_2 | 2+⋯+ | x_n | 2）1/2
- ∞ - 范数：‖ x ‖ ∞ =max（| x_1 |，| x_2 |，⋯，| x_n |）

其中 2- 范数就是通常意义下的距离。

以下示例演示如何加载 libsvm 格式的数据集，然后将每行标准化为单位 L1 范数和单位 L ∞范数。

```
import org.apache.spark.ml.feature.Normalizer
import org.apache.spark.ml.linalg.Vectors

val dataFrame = spark.createDataFrame(Seq(
  (0, Vectors.dense(1.0, 0.5, -1.0)),
  (1, Vectors.dense(2.0, 1.0, 1.0)),
  (2, Vectors.dense(4.0, 10.0, 2.0))
)).toDF("id", "features")

// 使用 L^1 正规化向量
val normalizer = new Normalizer()
  .setInputCol("features")
  .setOutputCol("normFeatures")
  .setP(1.0)

val l1NormData = normalizer.transform(dataFrame)
```

```
println("Normalized using L^1 norm")
l1NormData.show()

// 使用 L^∞正规化向量
val lInfNormData = normalizer.transform(dataFrame, normalizer.p -> Double.
PositiveInfinity)
println("Normalized using L^inf norm")
lInfNormData.show()
```

4.2.14 规范化

规范化（StandardScaler）是转换向量行的数据集，标准化每个特征使得其有统一的标准差或者均值为零，方差为 1。Spark 中可以选择是带或者不带均值和方差，参数如下：

- withStd：默认值为真，使用统一标准差方式。
- withMean：默认为假。此种方法将产出一个稠密输出，所以应用于稀疏输入时要小心。

StandardScaler 是一个 Estimator，它可以使数据集产生一个 StandardScalerModel，用来计算汇总统计。然后，产生的模可以用来转换向量至统一的标准差或者零均值特征。注意，如果特征的标准差为零，则该特征在向量中返回的默认值为 0.0。

以下示例演示如何以 libsvm 格式加载数据集，然后规范化每个特征的单位标准偏差。

```
import org.apache.spark.ml.feature.StandardScaler

val dataFrame = spark.read.format("libsvm").load("file:///u01/bigdata/spark/
                data/mllib/sample_libsvm_data.txt")

val scaler = new StandardScaler()
  .setInputCol("features")
  .setOutputCol("scaledFeatures")
  .setWithStd(true)
  .setWithMean(false)

// 通过拟合 StandardScaler 来计算汇总统计
val scalerModel = scaler.fit(dataFrame)

// 规范每个特征使其有单位标准偏差
val scaledData = scalerModel.transform(dataFrame)
scaledData.show()
```

4.2.15 最大值—最小值缩放

最大值—最小值缩放（MinMaxScaler）通过重新调节大小将 Vector 形式的行转换到指定的范围内，通常为 [0,1]，它的参数有：

- min：默认为 0.0，为转换后所有特征的下边界。
- max：默认为 1.0，为转换后所有特征的上边界。

MinMaxScaler 计算数据集的汇总统计量，并产生一个 MinMaxScalerModel。该模型可以将独立的特征的值转换到指定的范围内。对于特征 E 来说，调整后的特征值如下：

$$Rescaled(e_i)=\frac{e_i-E_{\min}}{E_{\max}-E_{\min}}*(\max-\min)+\min$$

如果 $E_{max}==E_{min}$，则 $Rescaled=0.5*(max-min)$。

注意，因为零值转换后可能变为非零值，所以即便为稀疏输入，输出也可能为稠密向量。

下面的示例展示了如何读入一个 libsvm 形式的数据以及调整其特征值到 [0,1] 之间。

```
import org.apache.spark.ml.feature.MinMaxScaler
import org.apache.spark.ml.linalg.Vectors

val dataFrame = spark.createDataFrame(Seq(
  (0, Vectors.dense(1.0, 0.1, -1.0)),
  (1, Vectors.dense(2.0, 1.1, 1.0)),
  (2, Vectors.dense(3.0, 10.1, 3.0))
)).toDF("id", "features")

val scaler = new MinMaxScaler()
  .setInputCol("features")
  .setOutputCol("scaledFeatures")

// 计算汇总统计并生成 MinMaxScalerModel
val scalerModel = scaler.fit(dataFrame)

// 将每个特征重新缩放至 [min,max] 范围
val scaledData = scalerModel.transform(dataFrame)
println(s"Features scaled to range: [${scaler.getMin}, ${scaler.getMax}]")
scaledData.select("features", "scaledFeatures").show()
```

显示结果如下：

```
Features scaled to range: [0.0, 1.0]
+--------------+--------------+
|      features|scaledFeatures|
+--------------+--------------+
|[1.0,0.1,-1.0]| [0.0,0.0,0.0]|
| [2.0,1.1,1.0]| [0.5,0.1,0.5]|
|[3.0,10.1,3.0]| [1.0,1.0,1.0]|
+--------------+--------------+
```

4.2.16　最大值—绝对值缩放

最大值—绝对值缩放（MaxAbsScaler）转换 Vector 行的数据集，通过划分每个特征的最大绝对值，将每个特征重新缩放到 [-1，1] 中。它不会移动 / 居中数据，因此不会破坏任何稀疏性。MaxAbsScaler 计算数据集上的汇总统计信息，并生成 MaxAbsScalerModel。然后，模型可以将每个特征单独缩放到 [-1，1] 之间。

以下示例演示如何加载 libsvm 格式的数据集，然后将每个特征重新缩放到 [-1，1]。

```
import org.apache.spark.ml.feature.MaxAbsScaler
import org.apache.spark.ml.linalg.Vectors

val dataFrame = spark.createDataFrame(Seq(
  (0, Vectors.dense(1.0, 0.1, -8.0)),
  (1, Vectors.dense(2.0, 1.0, -4.0)),
  (2, Vectors.dense(4.0, 10.0, 8.0))
)).toDF("id", "features")

val scaler = new MaxAbsScaler()
  .setInputCol("features")
  .setOutputCol("scaledFeatures")

// 计算汇总统计并生成 MaxAbsScalerModel
val scalerModel = scaler.fit(dataFrame)

// 将每个特征缩放到 [-1,1]
val scaledData = scalerModel.transform(dataFrame)
scaledData.select("features", "scaledFeatures").show()
```

运行结果如下：

```
+--------------+----------------+
|      features|  scaledFeatures|
+--------------+----------------+
|[1.0,0.1,-8.0]|[0.25,0.01,-1.0]|
|[2.0,1.0,-4.0]|  [0.5,0.1,-0.5]|
|[4.0,10.0,8.0]|   [1.0,1.0,1.0]|
+--------------+----------------+
```

4.2.17 离散化重组

离散化重组（Bucketizer）是将一列连续特征转换为一列特征桶，其中的桶由用户指定。它涉及的参数如下：

- split：用于将连续特征映射到 bucket。对于 $n+1$ 分割，有 n 个桶。由分割 x，y 定义的桶保存除了最后一个桶之外的范围 [x，y）中的值，其还包括 y。分割应严格增加。必须明确提供（-inf，inf）的值以涵盖所有 Double 值；否则，指定拆分之外的值将被视为错误。分割的两个示例是 Array（Double.NegativeInfinity，0.0,1.0，Double.PositiveInfinity）和 Array（0.0,1.0,2.0）。

注意，如果你不知道目标列的上限和下限，你应该添加 Double.NegativeInfinity 和 Double.PositiveInfinity 作为分割的边界，以防止潜在的 Bucketizer 边界异常。还要注意，你提供的分割必须严格按升序，即 s0 <s1 <s2 < … <sn。更多详细信息，请参阅 Bucketizer 的 API 文档。

以下示例演示如何将双列存储到另一个索引列的列中。

```
import org.apache.spark.ml.feature.Bucketizer

val splits = Array(Double.NegativeInfinity, -0.5, 0.0, 0.5, Double.
        PositiveInfinity)

val data = Array(-999.9, -0.5, -0.3, 0.0, 0.2, 999.9)
val dataFrame = spark.createDataFrame(data.map(Tuple1.apply)).toDF("features")

val bucketizer = new Bucketizer()
  .setInputCol("features")
  .setOutputCol("bucketedFeatures")
  .setSplits(splits)

// 将原来的数值转换为箱式索引
val bucketedData = bucketizer.transform(dataFrame)

println(s"Bucketizer output with ${bucketizer.getSplits.length-1} buckets")
bucketedData.show()
```

运行结果如下：

```
+--------+----------------+
|features|bucketedFeatures|
+--------+----------------+
|  -999.9|             0.0|
|    -0.5|             1.0|
|    -0.3|             1.0|
|     0.0|             2.0|
|     0.2|             2.0|
|   999.9|             3.0|
+--------+----------------+
```

4.2.18　元素乘积

元素乘积（ElementwiseProduct）是使用元素级乘法将每个输入向量乘以提供的“权重”向量。换句话说，它通过标量乘法器来缩放数据集的每一列。这表示输入向量 $\boldsymbol{v}$ 和变换向量 $\boldsymbol{w}$ 之间的 Hadamard 乘积，以产生结果向量。

$$\begin{pmatrix} v_1 \\ \vdots \\ v_N \end{pmatrix} \circ \begin{pmatrix} w_1 \\ \vdots \\ w_N \end{pmatrix} = \begin{pmatrix} v_1 w_1 \\ \vdots \\ v_N w_N \end{pmatrix}$$

下面的示例演示了如何使用变换向量值来变换向量：

```
import org.apache.spark.ml.feature.ElementwiseProduct
import org.apache.spark.ml.linalg.Vectors

// 创建一些向量数据，也适用于稀疏向量
```

```
val dataFrame = spark.createDataFrame(Seq(
  ("a", Vectors.dense(1.0, 2.0, 3.0)),
  ("b", Vectors.dense(4.0, 5.0, 6.0)))).toDF("id", "vector")

val transformingVector = Vectors.dense(0.0, 1.0, 2.0)
val transformer = new ElementwiseProduct()
  .setScalingVec(transformingVector)
  .setInputCol("vector")
  .setOutputCol("transformedVector")

// 创建新列，可批量转换向量
transformer.transform(dataFrame).show()
```

运行结果如下：

```
+---+-------------+-----------------+
| id|       vector|transformedVector|
+---+-------------+-----------------+
|  a|[1.0,2.0,3.0]|    [0.0,2.0,6.0]|
|  b|[4.0,5.0,6.0]|   [0.0,5.0,12.0]|
+---+-------------+-----------------+
```

4.2.19 SQL 转换器

SQL 转换器（SQLTransformer）可以实现由 SQL 语句定义的变换。目前我们只支持 SQL 语法，如“SELECT … FROM __THIS__ …”，其中“__THIS__”表示输入数据集的基础表。select 子句指定要在输出中显示的字段、常量和表达式，并且可以是 Spark SQL 支持的任何 select 子句。用户还可以使用 Spark SQL 内置函数和 UDF 操作这些选定的列。例如，SQLTransformer 支持以下语句：

- SELECT a, a + b AS a_b FROM __THIS__
- SELECT a, SQRT(b) AS b_sqrt FROM __THIS__ where a > 5
- SELECT a, b, SUM(c) AS c_sum FROM __THIS__ GROUP BY a, b

假设我们有以下 DataFrame，它包含 id、v1 和 v2 三列：

```
id  |  v1 |  v2
----|-----|-----
 0  | 1.0 | 3.0
 2  | 2.0 | 5.0
```

这是 SQL 转换器“SELECT *, (v1 + v2) AS v3, (v1 * v2) AS v4 FROM __THIS__”：语句的输出。

```
id  |  v1 |  v2 |  v3 |  v4
----|-----|-----|-----|-----
 0  | 1.0 | 3.0 | 4.0 | 3.0
 2  | 2.0 | 5.0 | 7.0 |10.0
```

以下是实现以上结果的具体代码：

```
import org.apache.spark.ml.feature.SQLTransformer

val df = spark.createDataFrame(
  Seq((0, 1.0, 3.0), (2, 2.0, 5.0))).toDF("id", "v1", "v2")

val sqlTrans = new SQLTransformer().setStatement(
  "SELECT *, (v1 + v2) AS v3, (v1 * v2) AS v4 FROM __THIS__")

sqlTrans.transform(df).show()
```

4.2.20　向量汇编

向量汇编（VectorAssembler）是一个变换器，可以将给定的多列表组合成一个单一的向量列。它多用于将由不同特征变换器生成的原始特征和特征组合成单个特征向量，以便训练 ML 模型，如逻辑回归和决策树。VectorAssembler 接受的输入列类型为：所有数字类型，布尔类型和向量类型。在每一行中，输入列的值将按指定的顺序连接到向量中。

假设我们有一个带有 id、hour、mobile、userFeatures 和 clicked 五列的 DataFrame：

```
 id | hour | mobile | userFeatures     | clicked
----|------|--------|------------------|---------
 0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0
```

userFeatures 是一个包含三个用户特征的向量列。我们希望将 hour、mobile 和 userFeatures 合并成一个称为特征的单一特征向量，并使用它来预测是否被点击。如果我们将 VectorAssembler 的输入列设置为 hour、mobile 和 userFeatures，并将列输出到特征，则在转换后，我们应该得到以下 DataFrame：

```
 id | hour | mobile | userFeatures     | clicked | features
----|------|--------|------------------|---------|-----------------------------
 0  | 18   | 1.0    | [0.0, 10.0, 0.5] | 1.0     | [18.0, 1.0, 0.0, 10.0, 0.5]
```

以下是实现上述功能的代码：

```
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.linalg.Vectors

val dataset = spark.createDataFrame(
  Seq((0, 18, 1.0, Vectors.dense(0.0, 10.0, 0.5), 1.0))
).toDF("id", "hour", "mobile", "userFeatures", "clicked")

val assembler = new VectorAssembler()
  .setInputCols(Array("hour", "mobile", "userFeatures"))
  .setOutputCol("features")

val output = assembler.transform(dataset)
```

```
println("Assembled columns 'hour', 'mobile', 'userFeatures' to vector column
'features'")
output.select("features", "clicked").show(false)
```

4.2.21 分位数离散化

分位数离散化（QuantileDiscretizer）是将具有连续功能的列转换为带有分级分类功能的列。分级数量由参数 numBuckets 设置。

NaN 值：在 QuantileDiscretizer 拟合过程中，NaN 值将从列中移除。这将产生一个 Bucketizer 模型进行预测。在转换过程中，Bucketizer 会在数据集中找到 NaN 值时引发错误，但用户也可以通过设置 handleInvalid 来选择保留或删除数据集中的 NaN 值。如果用户选择保留 NaN 值，那么它们将被特别处理并放入自己的桶中，例如，如果使用 4 个桶，那么非 NaN 数据将被放入桶 [0-3]，但是 NaN 将是计数在特殊的桶 [4]。

算法：该分级范围使用近似算法（更多内容请自行查阅相关文档）。可以用参数来控制 relativeError 近似的精度。当设置为零时，计算精确的分位数（注意：计算精确分位数是一项昂贵的操作）。分级的上下边界为负无穷到正无穷，覆盖所有的实数值。

假设我们有如下 DataFrame，包含 id、hour 两列：

```
id  | hour
----|------
 0  | 18.0
----|------
 1  | 19.0
----|------
 2  | 8.0
----|------
 3  | 5.0
----|------
 4  | 2.2
```

hour 是 Double 类型的连续特征。我们希望将连续特征变成一个分级特征。给定 numBuckets = 3，可得到以下 DataFrame：

```
id  | hour | result
----|------|------
 0  | 18.0 | 2.0
----|------|------
 1  | 19.0 | 2.0
----|------|------
 2  | 8.0  | 1.0
----|------|------
 3  | 5.0  | 1.0
----|------|------
 4  | 2.2  | 0.0
```

实现以上功能的 Scala 代码如下：

```
import org.apache.spark.ml.feature.QuantileDiscretizer

val data = Array((0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.2))
val df = spark.createDataFrame(data).toDF("id", "hour")

val discretizer = new QuantileDiscretizer()
  .setInputCol("hour")
  .setOutputCol("result")
  .setNumBuckets(3)

val result = discretizer.fit(df).transform(df)
result.show()
```

4.3　特征选择

特征选择（Feature Selection）是从特征向量中选择那些更有效的特征，组成新的、更简单有效的特征向量的过程。它经常用于数据分析中，尤其用在高维数据分析中，可以剔除冗余或影响不大的特征，提升模型的性能。

4.3.1　向量机

向量机（VectorSlicer）是一个处理特征向量的变换器，会输出一个新的原始特征子集的特征向量。从向量列中提取特征很有用。VectorSlicer 接受具有指定索引的向量列，然后输出一个新的向量列，其值通过这些索引进行选择。有两种类型的索引：

- 代表向量中的索引的整数索引，setIndices()。
- 表示向量中特征名称的字符串索引，setNames()。

注意，向量列需要具有 AttributeGroup，因为要实现在 Attribute 的 name 字段匹配。整数和字符串的规格都可以接受。此外，你可以同时使用整数索引和字符串名称。但必须至少选择一个特征，重复的功能是不允许的，所以选择的索引和名称之间不能有重叠。
此外，如果选择了功能的名称，遇到空的输入属性时会抛出异常。
输出向量将首先（按照给定的顺序）对所选索引的特征进行排序，其次是所选择的名称（按照给定的顺序）。

假设我们有一个 DataFrame 与列 userFeatures：

```
userFeatures
------------------
 [0.0, 10.0, 0.5]
```

userFeatures 是一个包含三个用户特征的向量列。假设 userFeatures 的第一列全部为零，

因此我们要删除它并仅选择最后两列。VectorSlicer 使用 setIndices(1,2) 选择最后两个元素，然后生成一个名为 features 的新向量列：

```
 userFeatures     | features
------------------|-----------------------------
 [0.0, 10.0, 0.5] | [10.0, 0.5]
```

假设 userFeatures 有输入属性，如 [“f1”，“f2”，“f3”]，那么我们可以使用 setNames(“f2”，“f3”）来选择它们。

```
     userFeatures     | features
    ------------------|-----------------------------
     [0.0, 10.0, 0.5] | [10.0, 0.5]
["f1", "f2", "f3"] | ["f2", "f3"]
```

以下是实现向量选择的一个 Scala 代码示例：

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.types.StructType

val data = Arrays.asList(
  Row(Vectors.sparse(3, Seq((0, -2.0), (1, 2.3)))),
  Row(Vectors.dense(-2.0, 2.3, 0.0))
)

val defaultAttr = NumericAttribute.defaultAttr
val attrs = Array("f1", "f2", "f3").map(defaultAttr.withName)
val attrGroup = new AttributeGroup("userFeatures", attrs.asInstanceOf[Array[Attribute]])

val dataset = spark.createDataFrame(data, StructType(Array(attrGroup.toStructField())))

val slicer = new VectorSlicer().setInputCol("userFeatures").
            setOutputCol("features")

slicer.setIndices(Array(1)).setNames(Array("f3"))
// 也可以用 slicer.setIndices(Array(1, 2)) 或者 slicer.setNames(Array("f2", "f3"))

val output = slicer.transform(dataset)
output.show(false)
```

运行结果：

```
+--------------------+-------------+
|userFeatures        |features     |
+--------------------+-------------+
|(3,[0,1],[-2.0,2.3])|(2,[0],[2.3])|
|[-2.0,2.3,0.0]      |[2.3,0.0]    |
+--------------------+-------------+
```

4.3.2　R 公式

R 公式（RFormula）可以通过 R 模型公式来将数据中的字段转换为特征值。目前，ML 支持 R 运算符的有限子集，“~”“.”“:”“+”和“-”。基本操作有：

- “~”用于分隔目标和对象。
- “+”，连接词，“+ 0”表示删除空格。
- “-”，删除术语，“-1”表示删除空格。
- “:”用于交互（数字乘法或类别二元化）。
- “.”用于除目标外的所有列。

假设 a 和 b 为 double 数据类型的两列：

1）y ~ a + b 表示模型 y ~ w0 + w1 * a +w2 * b，其中 w0 为截距，w1 和 w2 为相关系数。

2）y ~a + b + a:b - 1 表示模型 y ~ w1* a + w2 * b + w3 * a * b，其中 w1、w2、w3 是相关系数。

RFormula 产生一个特征向量和一个 double 或者字符串标签列（label）。像 R 在线性回归中使用公式时，字符串输入列将被单编码，数字列将被转换为双精度。如果类别列是字符串类型，它将通过 StringIndexer 转换为 double 类型索引。如果标签列不存在，则输出中将通过规定的响应变量创造一个标签列。

假设我们有一个含有 id、country、hour 和 clicked 四列的 DataFrame：

```
id | country |hour  | clicked
---|---------|------|---------
 7 | "US"    | 18   | 1.0
 8 | "CA"    | 12   | 0.0
 9 | "NZ"    | 15   | 0.0
```

如果我们使用 RFormula 公式 clicked ~ country+ hour，则表明我们希望基于 country 和 hour 预测 clicked，通过转换我们可以得到如下 DataFrame：

```
id | country |hour  | clicked | features          | label
---|---------|------|---------|-------------------|-------
 7 | "US"    | 18   | 1.0     | [0.0, 0.0, 18.0] | 1.0
 8 | "CA"    | 12   | 0.0     | [0.0, 1.0, 12.0] | 0.0
 9 | "NZ"    | 15   | 0.0     | [1.0, 0.0, 15.0] | 0.0
```

以下是实现上述功能的 Scala 代码：

```
import org.apache.spark.ml.feature.RFormula

val dataset = spark.createDataFrame(Seq(
  (7, "US", 18, 1.0),
  (8, "CA", 12, 0.0),
  (9, "NZ", 15, 0.0)
)).toDF("id", "country", "hour", "clicked")
```

```
val formula = new RFormula()
  .setFormula("clicked ~ country + hour")
  .setFeaturesCol("features")
  .setLabelCol("label")

val output = formula.fit(dataset).transform(dataset)
output.select("features", "label").show()
```

4.3.3 卡方特征选择

卡方特征选择（ChiSqSelector）代表卡方特征选择。它适用于带有类别特征的标签数据。ChiSqSelector 根据分类的卡方独立性检验来对特征排序，选取类别标签主要依赖的特征。它类似于选取最有预测能力的特征。ML 目前支持三种选择方法：numTopFeatures、percentile 和 fpr。

- numTopFeatures：根据卡方测试选择固定数量的排名靠前的几个特征。这类似于产生具有最大预测能力的特征。
- percentile：类似于 numTopFeatures 但选择所有特征的一部分，而不是固定数量。
- fpr：选择 p 值低于阈值的所有特征，从而控制选择的假阳性率。

默认情况下，使用 numTopFeatures 将默认的排名数量设置为 50。用户可以使用 setSelectorType 选择方法。

假设我们有一个含有 id,features 和 clicked 三列的 DataFrame，其中 clicked 为需要预测的目标：

```
id | features              | clicked
---|-----------------------|---------
 7 | [0.0, 0.0, 18.0, 1.0] | 1.0
 8 | [0.0, 1.0, 12.0, 0.0] | 0.0
 9 | [1.0, 0.0, 15.0, 0.1] | 0.0
```

如果我们使用 ChiSqSelector 并设置 numTopFeatures 为 1，根据标签 clicked，features 中最后一列将会是最有用的特征：

```
id | features              | clicked | selectedFeatures
---|-----------------------|---------|------------------
 7 | [0.0, 0.0, 18.0, 1.0] | 1.0     | [1.0]
 8 | [0.0, 1.0, 12.0, 0.0] | 0.0     | [0.0]
 9 | [1.0, 0.0, 15.0, 0.1] | 0.0     | [0.1]
```

下面是使用 ChiSqSelector 的 Scala 代码示例：

```
import org.apache.spark.ml.feature.ChiSqSelector
import org.apache.spark.ml.linalg.Vectors

val data = Seq(
  (7, Vectors.dense(0.0, 0.0, 18.0, 1.0), 1.0),
  (8, Vectors.dense(0.0, 1.0, 12.0, 0.0), 0.0),
```

```
    (9, Vectors.dense(1.0, 0.0, 15.0, 0.1), 0.0)
  )

  val df = spark.createDataset(data).toDF("id", "features", "clicked")

  val selector = new ChiSqSelector()
    .setNumTopFeatures(1)
    .setFeaturesCol("features")
    .setLabelCol("clicked")
    .setOutputCol("selectedFeatures")

  val result = selector.fit(df).transform(df)

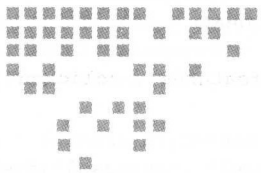
  println(s"ChiSqSelector output with top ${selector.getNumTopFeatures} features
selected")
  result.show()
```

结果显示：

```
+---+------------------+-------+----------------+
| id|          features|clicked|selectedFeatures|
+---+------------------+-------+----------------+
|  7|[0.0,0.0,18.0,1.0]|    1.0|          [18.0]|
|  8|[0.0,1.0,12.0,0.0]|    0.0|          [12.0]|
|  9|[1.0,0.0,15.0,0.1]|    0.0|          [15.0]|
+---+------------------+-------+----------------+
```

4.4　小结

本章主要介绍了对数据特征或变量的一些常用操作，包括特征提取、特征转换以及特征选择等方法，这些任务在实际项目中往往会花费大量时间和精力，尤其要自己编写这方面的代码或函数时更是如此，Spark ML 目前提供了很多现成的函数，有效使用这些函数将有助于提供我们的开发效率，同时使我们有更多时间优化或提升模型性能。下一章我们将介绍优化或提升模型性能的一些方法。



Chapter 5 第 5 章

模型选择和优化

本章主要介绍如何使用 Spark ML 提供的方法及自定义函数等方法来对模型进行调优。我们可以通过 Spark ML 内建的交叉验证、训练验证拆分法、网格参数等方法进行模型调优，当然也可以自定义函数进行模型优化。

本章主要内容包括：

- ❑ 模型选择
- ❑ 交叉验证
- ❑ 训练验证拆分法
- ❑ 自定义函数调优

5.1 模型选择

调优可以对单个 Estimator 进行，比如逻辑回归，或者对包含多个算法、特征化和其他步骤的整个 Pipline 进行。用户可以一次性对整个 Pipline 进行调优，而不必对 Pipline 中的每一个元素进行单独调优。

MLlib 支持使用像交叉验证（CrossValidator）和训练验证拆分法（TrainValidationSplit）这样的工具进行模型选择（Model Selection）。这些工具需要以下组件：

- ❑ Estimator：用户调优的算法或 Pipeline。
- ❑ ParamMap 集合：提供参数选择，有时也叫作用户查找的“参数网格”。
- ❑ Evaluator：衡量模型在测试数据上的拟合程度。

这些模型选择工具的工作方式如下：

- 将输入数据切分成训练数据集和测试数据集。
- 对于每一个（训练数据，测试数据）对，通过 ParamMap 集合进行迭代：
 - 对于每个 ParamMap，使用它提供的参数对 Estimator 进行拟合，给出拟合模型，然后使用 Evaluator 来评估模型的性能。
- 选择表现最好的参数集合生成的模型。

其中，对于回归问题评估器可选择 RegressionEvaluator，二值数据可选择 BinaryClassifica-tionEvaluator，多分类问题可选择 MulticlassClassificationEvaluator。评估器里默认的评估准则可通过 setMetricName 方法重写。用户可通过 ParamGridBuilder 构建参数网格。

5.2　交叉验证

模型选择是 Spark ML 的重要内容，而采用 Pipeline 方式又使模型选择变得可行高效。调优时，只需对整个 Pipeline 进行调整，而不必对每个元素调整。目前 ML 使用交叉验证（Cross-validation）进行模型选择。Cross-validator 本身带有一个 Estimator、一个 Evaluator 及一组 ParamMap。

Cross-validator 先将数据集划分多组（如 k 组），每组由训练集和测试集组成。例如，k=3 折时，CrossValidator 会生成 3 个（训练数据，测试数据）对，每一个数据对的训练数据占 2/3，测试数据占 1/3。迭代 ParamMap 集合，CrossValidator 会计算这三个不同的（训练，测试）数据集对在 Estimator 拟合出的模型上的平均评估指标。在找出最好的 ParamMap 后，CrossValidator 会使用这个 ParamMap 和整个数据集来重新拟合 Estimator。

在整个参数网格中进行交叉验证是比较耗时的。例如，在下面的例子中，参数网格有 3 个 hashingTF.numFeatures 值和 2 个 lr.regParam 值，CrossValidator 使用 2 折切分数据。最终将有（3 * 2）* 2 = 12 个不同的模型将被训练。在真实场景中，很可能使用更多的参数和进行更多折切分（k=3 和 k=10 都很常见）。使用 CrossValidator 的代价可能会异常的高，当数据集比较大时，需要慎重选择。不过采用交叉验证法，对比手动调优，还是有较大优势。

下面通过示例说明如何使用 CrossValidator 从整个网格的参数中选择合适的参数。

1）导入必要的包：

```
import org.apache.spark.ml.classification.{LogisticRegression,LogisticRegressionM
odel}
import org.apache.spark.ml.{Pipeline,PipelineModel}
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.linalg.Vector
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.sql.Row
```

2）生成一个含 id、text、label 的训练数据集：

```
val training = spark.createDataFrame(Seq(
  (0L, "a b c d e spark", 1.0),
  (1L, "b d", 0.0),
  (2L, "spark f g h", 1.0),
  (3L, "hadoop mapreduce", 0.0),
  (4L, "b spark who", 1.0),
  (5L, "g d a y", 0.0),
  (6L, "spark fly", 1.0),
  (7L, "was mapreduce", 0.0),
  (8L, "e spark program", 1.0),
  (9L, "a e c l", 0.0),
  (10L, "spark compile", 1.0),
  (11L, "hadoop software", 0.0)
)).toDF("id", "text", "label")
```

3）配置一个流水线，该流水线包含 3 个 stage: tokenizer、hashingTF 和 lr。

```
val tokenizer = new Tokenizer()
  .setInputCol("text")
  .setOutputCol("words")
val hashingTF = new HashingTF()
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression()
  .setMaxIter(10)
val pipeline = new Pipeline()
  .setStages(Array(tokenizer, hashingTF, lr))
```

4）使用 ParamGridBuilder 构造一个参数网格：

```
val paramGrid = new ParamGridBuilder()
  .addGrid(hashingTF.numFeatures, Array(10, 100, 1000))
  .addGrid(lr.regParam, Array(0.1, 0.01))
  .build()
```

5）将流水线嵌入到 CrossValidator 实例中，这样流水线的任务都可使用参数网格。BinaryClassificationEvaluator 默认的评估指标为 AUC(areaUnderROC)。

```
val cv = new CrossValidator()
  .setEstimator(pipeline)
  .setEvaluator(new BinaryClassificationEvaluator)
  .setEstimatorParamMaps(paramGrid)
  .setNumFolds(2)
```

6）通过交叉验证模型，获取最优参数集，并测试模型：

```
val cvModel = cv.fit(training)
val test = spark.createDataFrame(Seq(
  (4L, "spark i j k"),
  (5L, "l m n"),
  (6L, "mapreduce spark"),
```

```
    (7L, "apache hadoop")
)).toDF("id", "text")

val Predictions=cvModel.transform(test)
Predictions.select("id", "text", "probability", "prediction")
  .collect()
  .foreach { case Row(id: Long, text: String, prob: Vector, prediction: Double)
=>
    println(s"($id, $text) --> prob=$prob, prediction=$prediction")
  }
```

7）查看最佳模型中的各参数值：

```
val bestModel= cvModel.bestModel.asInstanceOf[PipelineModel]
val lrModel = bestModel.stages(2).asInstanceOf[LogisticRegressionModel]

lrModel.getRegParam  //显示为 :0.1
lrModel.numFeatures  //显示结果为 :100
```

5.3　训练验证拆分法

除 CrossValidator 外，Spark 还提供了用于超参数调优的训练验证拆分法（TrainValidationSplit）。不同于 CrossValidator，TrainValidationSplit 创建单一的（训练，测试）数据集对。相对于 CrossValidator 的 k 次评估，TrainValidationSplit 只对每个参数组合评估一次。因此它的评估代价没有那么高，但在训练数据集不够大时其结果可能没有交叉验证的理想。TrainValidationSplit 将生成训练和测试数据集对，一般是 75% 的数据用于培训，25% 用于验证，当然，这个可以根据需要进行修改或调整。

像 CrossValidator 一样，TrainValidationSplit 最终适合使用最好的 ParamMap 和整个数据集的 Estimator。

```
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.regression.LinearRegression
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}

// 准备训练数据和测试数据
val data = spark.read.format("libsvm").load("file:///u01/bigdata/spark/data/
      mllib/sample_linear_regression_data.txt")
val Array(training, test) = data.randomSplit(Array(0.9, 0.1), seed = 12345)

val lr = new LinearRegression()
    .setMaxIter(10)

// 我们使用 ParamGridBuilder 构建一个搜索参数网格
// TrainValidationSplit 将尝试所有值的组合，并确定使用最佳模型
val paramGrid = new ParamGridBuilder()
```

```
  .addGrid(lr.regParam, Array(0.1, 0.01))
  .addGrid(lr.fitIntercept)
  .addGrid(lr.elasticNetParam, Array(0.0, 0.5, 1.0))
  .build()

// 在这种情况下，估计量就是线性回归
// TrainValidationSplit 需要估计器，一组估计器参数映射和一个评估器
val trainValidationSplit = new TrainValidationSplit()
  .setEstimator(lr)
  .setEvaluator(new RegressionEvaluator)
  .setEstimatorParamMaps(paramGrid)
  .setTrainRatio(0.8)
// 80% 的数据用于培训，其余 20% 用于验证

// 使用验证拆分训练，并选择最佳的一组参数
val model = trainValidationSplit.fit(training)

// 对测试数据进行预测，模型具有最佳性能的参数组合
model.transform(test)
  .select("features", "label", "prediction")
  .show()
```

5.4 自定义模型选择

下面，我们将通过一个完整示例来讲解自定义模型选择。

```
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.Row
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.DataFrame
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.recommendation.{ALS,ALSModel}

case class Rating(userId: Int, movieId: Int, rating: Float, timestamp: Long)

def parseRating(str: String): Rating = {
  val fields = str.split("::")
  assert(fields.size == 4)
  Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat, fields(3).toLong)
}

val ratings = spark.read.textFile("file:///u01/bigdata/spark/data/mllib/als/
              sample_movielens_ratings.txt")
  .map(parseRating)
  .toDF()

// 将样本评分表分成 3 个部分，分别用于训练 (60%)，校验 (20%) 和测试 (20%)
```

```
val splits = ratings.randomSplit(Array(0.6, 0.2,0.2),12)
//把训练样本缓存起来，加快运算速度
val training = splits(0).cache()
val validation = splits(1).toDF.cache()
val test = splits(1).toDF.cache()
//计算各集合总数
val numTraining = training.count()
val numValidation = validation.count()
val numTest = test.count()

//训练不同参数下的模型，并在校验集中验证，获取最佳参数下的模型
val ranks = List(20, 40)
val lambdas = List(0.01, 0.1)
val numIters = List(5, 10)
var bestModel: Option[ALSModel] = None
var bestValidationRmse = Double.MaxValue
var bestRank = 0
var bestLambda = 1.0
var bestNumIter = 1

def computeRmse(model:ALSModel,data:DataFrame,n:Long):Double = {
    val predictions = model.transform(data)
    val p1=predictions.rdd.map{ x =>((x(0),x(1)),x(2))}.join(predictions.rdd.
          map{ x =>((x(0),x(1)),x(4))}).values
    math.sqrt(p1.map( x => (x._1.toString.toDouble - x._2.toString.toDouble) *
(x._1.toString.toDouble - x._2.toString.toDouble)).reduce(_+_)/n)
  }

for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
val als = new ALS()
  .setMaxIter(numIter)
  .setRegParam(lambda)
  .setRank(rank)
  .setUserCol("userId")
  .setItemCol("movieId")
  .setRatingCol("rating")
val model = als.fit(training)
val validationRmse = computeRmse(model, validation, numValidation)
    println("RMSE(validation) = " + validationRmse + " for the model trained
                                  with rank = "
      + rank + ",lambda = " + lambda + ",and numIter = " + numIter + ".")
    if (validationRmse < bestValidationRmse) {
      bestModel = Some(model)
      bestValidationRmse = validationRmse
      bestRank = rank
      bestLambda = lambda
      bestNumIter = numIter
    }
  }
```

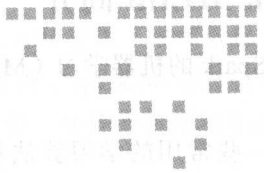
```
    RMSE(validation) = 1.544091979633677 for the model trained with rank = 20,lambda
= 0.01,and numIter = 5.
    RMSE(validation) = 1.3973269424767092 for the model trained with rank = 20,lambda
= 0.01,and numIter = 10.
    RMSE(validation) = 1.0381748242379452 for the model trained with rank = 20,lambda
= 0.1,and numIter = 5.
    RMSE(validation) = 1.0059139723438042 for the model trained with rank = 20,lambda
= 0.1,and numIter = 10.
    RMSE(validation) = 1.5545572907673506 for the model trained with rank = 40,lambda
= 0.01,and numIter = 5.
    RMSE(validation) = 1.390163394916565 for the model trained with rank = 40,lambda
= 0.01,and numIter = 10.
    RMSE(validation) = 1.0520138790334888 for the model trained with rank = 40,lambda
= 0.1,and numIter = 5.
    RMSE(validation) = 1.0132571682721812 for the model trained with rank = 40,lambda
= 0.1,and numIter = 10.

    //用最佳模型预测测试集的评分，并计算和实际评分之间的均方根误差(RMSE)
    val testRmse = computeRmse(bestModel.get, test, numTest)
    testRmse: Double = 1.0059139723438042

    println("The best model was trained with rank = " + bestRank + " and lambda = "
+ bestLambda + ", and numIter = " + bestNumIter + ", and its RMSE on the test set is
" + testRmse + ".")
    //最佳模型相关参数
    The best model was trained with rank = 20 and lambda = 0.1, and numIter = 10,
and its RMSE on the test set is 1.0059139723438042.
```

5.5 小结

本章主要介绍了几种模型选择或调优的方法，我们可以从训练的数据集入手，可以从模型参数入手，当然也可以把两者结合起来使用。实际上模型的优化还有很多其他方法，如使用不同的算法、集成算法等。下一章我们将介绍 Spark MLlib 的一些基础知识，包括 Spark MLlib 架构、原理、算法及算法依赖的一些库、向量和矩阵等相关内容。



第 6 章 Chapter 6

Spark MLlib 基础

传统的机器学习算法，由于技术和单机存储的限制，只能在少量数据上使用。一旦数据量过大，往往需要采用数据抽样的方法。但这种抽样很难保证不走样。近些年随着 HDFS 等分布式文件系统出现，存储海量数据已经成为可能。在全量数据上进行机器学习变得可能或必要，通过 MapReduce 计算框架虽然可以实现分布式计算，但中间结果需要存在到磁盘，这对于计算过程中需要多次迭代的机器学习（因为通常情况下机器学习算法参数学习的过程都是迭代计算的）来说不很理想。

Spark 的出现正好弥补了 MapReduce 的不足，它立足于内存计算，所以特别适合机器学习的迭代式计算。同时 Spark 提供了一个基于海量数据的分布式运算的机器学习库，同时提供了很多特征选取、特征转换等内嵌函数，大大降低了大家学习和使用 Spark 的门槛。对开发者来说只需有一定 Spark 基础、了解机器学习算法的基本原理及相关参数的含义和作用，都可以比较顺利地使用 Spark 进行基于大数据的机器学习。

Spark 在机器学习方面有很多优势，本章主要介绍 Spark 与机器学习相关的内容。

- ❑ Spark MLlib 简介
- ❑ Spark MLlib 架构
- ❑ 常用的几种数据类型
- ❑ 基础统计
- ❑ RDD、DataFrame 及 Dataset 间的异同
- ❑ Spark MLlib 常用算法

6.1 Spark MLlib 简介

MLlib 是 Spark 的机器学习（ML）库，通过该库可简化机器学习的工程实践工作，并方便扩展。

MLlib 由一些常用的学习算法和工具组成，包括分类、回归、聚类、协同过滤、降维等，同时还包括底层的优化和高层的流水线 API。它基于 RDD，可以与 Spark SQL、GraphX、Spark Streaming 无缝连接。MLlib 目前分为两个代码包：

1. spark.mllib

Spark.mllib 包含基于 RDD 的原始算法 API。目前支持 4 种常见的机器学习问题：分类、回归、聚类和协同过滤。MLlib 在 Spark 整个生态系统中的位置如图 6-1 所示。

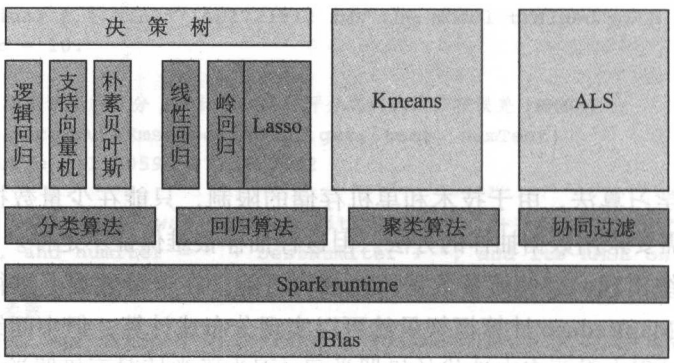


图 6-1 Spark MLlib 算法生态

MLlib 是 MLBase 一部分，其中 MLBase 分为四部分：MLlib、MLI、ML Optimizer 和 MLRuntime。它们的结构如图 6-2 所示。

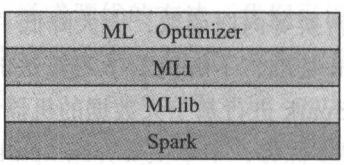


图 6-2 MLBase 四部分关系

- ML Optimizer：会选择它认为最适合的已经在内部实现好了的机器学习算法和相关参数，来处理用户输入的数据，并返回模型或别的帮助分析的结果；
- MLI：一个进行特征抽取和高级 ML 编程抽象的算法实现的 API 或平台；
- MLlib：提供了 Spark 实现一些常见的机器学习算法和实用程序，包括分类、回归、聚类、协同过滤、降维以及底层优化，该算法可以进行可扩充；
- Spark：MLRuntime 基于 Spark 计算框架，将 Spark 的分布式计算应用到机器学习领域。

2. spark.ml

提供了基于 DataFrames 高层次的 API，Spark ML API 提供了很多数据处理函数，如特征选取、特征转换、类别数值化、正则化、降维等。此外，spark.ml 通过构建一个机器学习流水线，为我们提供了更灵活的方法，把机器学习过程一些任务有序组织在一起，便于运行，也便于环境的迁移。Spark 官方推荐使用 spark.ml。

如果新的算法能够适用于机器学习管道的概念，就应该将其放到 spark.ml 包中，如特征提取器和转换器。开发者需要注意的是，从 Spark 2.0 开始，基于 RDD 的 API 进入维护模式（即不增加任何新的特性），并预期于 3.0 版本的时候被移除出 MLlib，因此，本书也主要以 ml 包为主。

6.2　Spark MLlib 架构

图 6-3 为 Spark MLlib 的架构图。

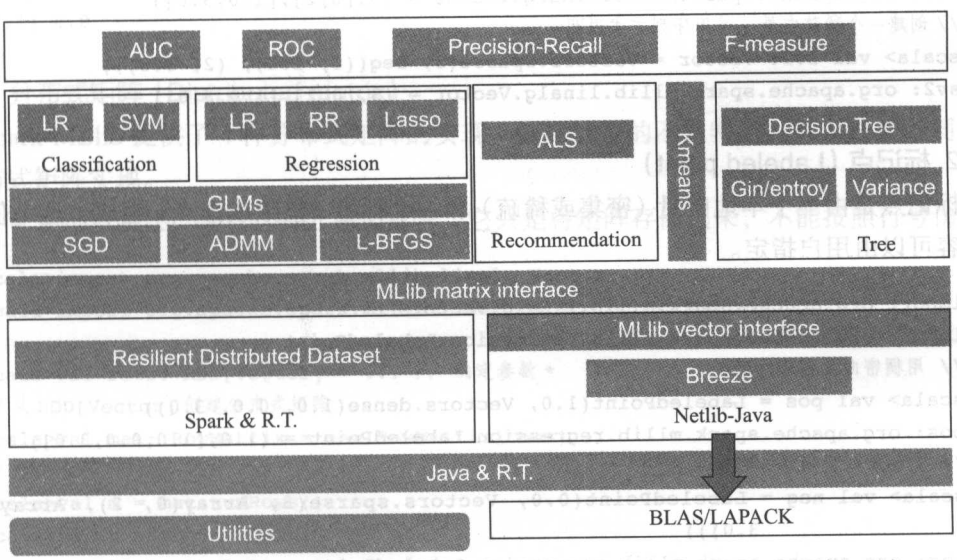


图 6-3　Spark MLlib 架构

由图 6-3 可知：

- 底层基础：包括 Spark 的运行库、矩阵库和向量库等。其中向量接口和矩阵接口都会使用 Scala 语言基于 Netlib 和 BLAS/LAPACK 开发的线性代数库 Breeze。MLlib 支持本地的密集向量和稀疏向量，并且支持标记向量。MLlib 同时支持本地矩阵和分布式矩阵，支持的分布式矩阵分为 RowMatrix、IndexedRowMatrix、CoordinateMatrix 等。
- 算法库：包含广义线性模型、推荐系统、聚类、决策树和评估的算法。
- 模型评级：对模型进行评估。

6.3 数据类型

Spark MLlib 的数据类型主要分为 4 种，下面将分别介绍。

1. 本地向量（Local vector）

本地向量主要分为 DenseVector 和 SparseVector 两种，前者用来保存稠密向量，后者用来保存稀疏向量，如向量（1.0,0.0,3.0）的稠密向量为 [1.0,0.0,3.0]，而对应的稀疏向量为（3, [0, 2], [1.0, 3.0]），此处，3 代表向量长度。其创建方式主要有以下几种：（以下使用 Scala 语言）

```
spark-shell --master spark://master:7077--total-executor-cores 2
// 创建一个稠密向量
scala>import org.apache.spark.mllib.linalg.{Vector, Vectors}
scala> val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
dv: org.apache.spark.mllib.linalg.Vector = [1.0,0.0,3.0]
// 创建一个稀疏向量，说明非零项的索引和值
scala> val sv1: Vector = Vectors.sparse(3, Array(0, 2), Array(1.0, 3.0))
sv1: org.apache.spark.mllib.linalg.Vector = (3,[0,2],[1.0,3.0])
// 创建一个稀疏向量，采用序列方式说明
scala> val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
sv2: org.apache.spark.mllib.linalg.Vector = (3,[0,2],[1.0,3.0])
```

2. 标记点（Labeled point）

标记点是由一个本地向量（密集或稀疏）和一个标签（整数或浮点）组成，这个值的具体内容可以由用户指定。

```
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.regression.LabeledPoint
// 用稠密向量创建标记点
scala> val pos = LabeledPoint(1.0, Vectors.dense(1.0, 0.0, 3.0))
pos: org.apache.spark.mllib.regression.LabeledPoint = (1.0,[1.0,0.0,3.0])
// 用稀疏向量创建标记点
scala> val neg = LabeledPoint(0.0, Vectors.sparse(3, Array(0, 2), Array(1.0,
                3.0)))
neg: org.apache.spark.mllib.regression.LabeledPoint = (0.0,(3,[0,2],[1.0,3.0]))
```

从文件中直接获取标记点：

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils
import org.apache.spark.rdd.RDD

val examples: RDD[LabeledPoint] = MLUtils.loadLibSVMFile(sc, "file:///home/
                                  hadoop/bigdata/spark/data/mllib/sample_libsvm_
                                  data.txt")
```

3. 本地矩阵（Local matrix）

由行索引、列索引、类型值组成，存放在单机中。

```
scala> import org.apache.spark.mllib.linalg.{Matrix, Matrices}
// 创建一个稠密矩阵((1.0, 2.0), (3.0, 4.0), (5.0, 6.0))
scala> val dm: Matrix = Matrices.dense(3, 2, Array(1.0, 3.0, 5.0, 2.0, 4.0, 6.0))
dm: org.apache.spark.mllib.linalg.Matrix =
1.0  2.0
3.0  4.0
5.0  6.0
// 创建一个稀疏矩阵((9.0, 0.0), (0.0, 8.0), (0.0, 6.0))
scala> val sm: Matrix = Matrices.sparse(3, 2, Array(0, 1, 3), Array(0, 2, 1),
Array(9, 6, 8))
sm: org.apache.spark.mllib.linalg.Matrix =
3 x 2 CSCMatrix
(0,0) 9.0
(2,1) 6.0
(1,1) 8.0
```

4. 分布式矩阵（Distributed matrix）

Spark MLlib 提供了 4 种分布式矩阵的实现，依据数据的不同特点，可以选择不同类型的分布式矩阵实现。

- RowMatrix：最基础的分布式矩阵，它只是将矩阵存储起来，不能按照行号访问。

```
scala>import org.apache.spark.mllib.linalg.Vector
scala>import org.apache.spark.mllib.linalg.distributed.RowMatrix

scala>val rows: RDD[Vector] = ... // 确定参数 *
//从 RDD[Vector] 创建分布式矩阵
scala>val mat: RowMatrix = new RowMatrix(rows)
// 获取矩阵行列数
scala>val m = mat.numRows()
scala>val n = mat.numCols()
```

如何得到一个 RDD[Vector]？以下是方法之一：

```
val v0 = Vectors.dense(1.0, 0.0, 3.0)
val v1 = Vectors.dense(3.0,2.0, 4.0)
val rows = sc.parallelize(Seq(v0, v1))    // 这是一个 RDD[Vector]
```

- IndexedRowMatrix：与 RowMatrix 矩阵的不同之处在于，它可以通过索引值来访问每一行，其他基本类似。
- CoordinateMatrix：当你的数据特别稀疏的时候怎么办？采用这种矩阵比较适合。
- BlockMatrix：分布式矩阵 RDD[MarixBlock]，此处 MatrixBlock 是元组（(Int, Int)，

Matrix），其中（Int, Int）是矩阵块的索引，Matrix 是给定矩阵块索引的子矩阵，矩阵维度（是数组的长度）为 rowsPerBlock * colsPerBlock。

6.4 基础统计

MLlib 提供了很多统计方法，是对 RDD 格式数据的统计，包括摘要统计、相关统计、分层统计、假设检验、随机数生成、流显著性验证、核密度估计等，这些涉及统计学、概率论等专业知识。下面就常用部分进行说明。

6.4.1 摘要统计

对于 RDD[Vector] 类型的列统计信息，Spark MLlib 提供了一种名为 colStats() 的统计方法，即摘要统计（summary statistic）方法。调用该方法可以计算每列的最大值，最小值，均值、方差、总数、L1 范数、L2 范数等。示例代码如下：

```
import org.apache.spark.mllib.linalg.{Vector,Vectors}
import org.apache.spark.mllib.stat.{MultivariateStatisticalSummary, Statistics}
val data=sc.parallelize(1 to 100,2)
val data1=data.map(x=>Vectors.dense(x))
val stat1=Statistics.colStats(data1)
// 可求 stat1 的如下统计值：
//count、max、mean、min、normL1、normL2、numNonzeros、variance 等，这里以 count 为例
stat1.count     // 显示为 100
stat1.normL1    // 显示为 5050.0
```

6.4.2 相关性

目前 Spark 支持两种相关性（correlation）系数：皮尔森相关系数（Pearson）和斯皮尔曼等级相关系数（Spearman）。相关系数是用以反映变量之间相关关系程度的统计指标。其取值范围是 [−1,1]，当取值为 0 时表示不相关，取值为 [−1 ～ 0) 表示负相关，取值为 (0, 1] 表示正相关。

Pearson 相关系数的计算公式：

$$r = r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

其中 n 为样本量。Pearson 相关系数反应两个数值变量的线性相关性，它一般适用于正态分布。

Spearman 相关系数也用来表达两个变量的相关性，但是它没有 Pearson 相关系数对变量分布的严格要求，其计算公式为：

$$\rho = 1 - \frac{6\sum_{i=1}^{n}(x_i - y_i)^2}{n(n^2-1)}$$

下面通过示例说明相关系统是如何计算的。

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.linalg._
import org.apache.spark.mllib.stat.Statistics

val x=sc.parallelize(Array(1.0,2.0,3.0,3.2,2.6))
val y=sc.parallelize(Array(4.0,5.0,6.0,4.2,5.6))

val corr_r=Statistics.corr(x, y, "pearson")
val corr_p=Statistics.corr(x, y, "spearman")
```

6.4.3　假设检验

关于假设检验（Hypothesis testing），Spark MLlib 目前支持皮尔森卡方检测（Pearson's chi-squared tests），包括适配度检测和独立性检测。适配度检测要求输入为 Vector，独立性检验要求输入是 Matrix。

- 适配度检测（Goodness of Fit test），验证一组观察值的次数分配是否异于理论上的分配。其 H0 假设（虚无假设，null hypothesis）为一个样本中已发生事件的次数分配会服从某个特定的理论分布。通常情况下这个特定的理论分配指的是均匀分布，目前 Spark 默认的是均匀分布。
- 独立性检测（independence test），验证从两个变量抽出的配对观察值组是否互相独立。其虚无假设是：两个变量呈统计独立性。

代码示例：

```
import org.apache.spark.mllib.linalg.{Matrix, Matrices, Vectors}
import org.apache.spark.mllib.stat.Statistics
import org.apache.spark.{SparkConf, SparkContext}

val vd = Vectors.dense(1, 2, 3, 4, 5)
val vdResult = Statistics.chiSqTest(vd)

val mtx = Matrices.dense(3, 2, Array(1, 3, 5, 2, 4, 6))
val mtxResult = Statistics.chiSqTest(mtx)
```

6.4.4　随机数据生成

RandomRDDs 是一个工具集，用来生成含有随机数的 RDD，可以按各种给定的分布模式（正态分布、泊松分布和均匀分布等）生成数据集，例如，按正态分布生成随机数。

代码示例：

```
import org.apache.spark.SparkContext
import org.apache.spark.mllib.random.RandomRDDs._

// 生成 100 个服从正态分布 N(0,1) 的 RDD〔Double〕，并且分布在 10 个分区中
val u = normalRDD(sc, 100L, 10)

// 把生成的随机数转换为 N(1,4) 正态分布
val v = u.map(x => 1.0 + 2.0 * x)
```

6.5 RDD、Dataframe 和 Dataset

目前，spark.mllib 包中基于 RDD 的 API 已进入维护模式，以后将以 spark.ml 包中的基于 DataFrame 的 API 为主。

这一改变，将会带来哪些影响？（以下主要信息摘自 Spark 官网：http://spark.apache.org。）

- MLlib 将继续支持基于 RDD 的 API。
- MLlib 不会向基于 RDD 的 API 中添加新的特征。
- 在 Spark 2.0 以后的版本中，将继续向基于 DataFrame 的 API 添加新特征以缩小与基于 RDD 的 API 的差异。
- 当两种接口之间达到特征相同时（初步估计为 Spark 2.2），不推荐使用基于 RDD 的 API。
- 基于 RDD 的 API 将在 Spark 3.0 中被移除。

MLlib 中基于 DataFrame 的 API 有哪些优势？

- DataFrames 提供比 RDD 更加用户友好的 API。DataFrames 包括 Spark Datasources、SQL/DataFrame 查询，采用 Tungsten 和 Catalyst 优化以及跨语言的统一 API。
- 基于 DataFrames 的 MLlib API 为多种 ML 算法与编程语言提供统一的接口。
- DataFrames 有助于实际的 ML Pipelines，特别是特征转换。

6.5.1 RDD

RDD 是 Spark 建立之初的核心 API，是一种有容错机制的特殊集合。RDD 是不可变分布式弹性数据集，在 Spark 集群中可跨节点分区，以函数式编程操作集合的方式进行各种并行操作，提供分布式 low-level API 来操作，包括 transformation 和 action 等。

它提供了一种只读、只能由已存在的 RDD 变换而来的共享内存，然后将所有数据都加载到内存中，方便进行多次重用。

RDD 优点：

- 编译时就能检查出类型错误。

❑ 面向对象的编程风格，可直接通过类名点的方式来操作数据。

RDD 缺点：

❑ 无论是集群间的通信，还是 IO 操作都需要对对象的结构和数据进行序列化和反序列化。

❑ 频繁地创建和销毁对象，势必会增加 GC 开销。

RDD 使用场景：

❑ 使用 low-level 的 transformation 和 action 来控制你的数据集。

❑ 使用非结构化数据集，比如，流媒体或者文本流等。

❑ 使用函数式编程来操作你的数据，而不是用特定领域语言（DSL）表达。

❑ 不在乎 schema，比如，当通过名字或者列处理（或访问）数据属性时，不在意列式存储格式。

❑ 不希望使用 DataFrame 和 Dataset 来优化结构化和半结构化数据集。

6.5.2　Dataset/DataFrame

DataFrame 与 RDD 都是不可变分布式弹性数据集。不同之处在于，DataFrame 多了数据的结构信息，即 schema，类似于传统数据库中的表。DataFrame 的设计是为了让大数据处理起来更容易，并做了 higher-level 的抽象。DataFrame 提供特定领域的语言（DSL）API 来操作数据集。

在 Spark 2.0 中，DataFrame API 将会和 Datasets API 合并，统一数据处理 API，如图 6-4 所示。

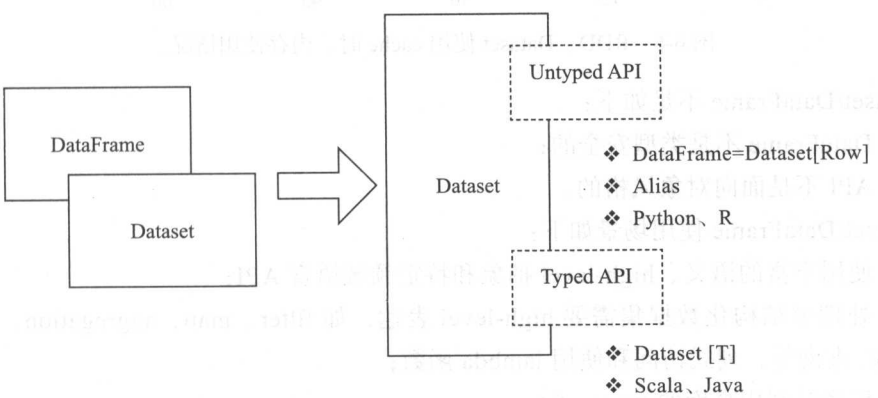


图 6-4　统一的 Apache Spark API

Dataset/DataFrame 优势如下：

（1）简单易用：虽然结构化数据会给 Spark 程序操作数据集带来一些限制，但它却引进了丰富的语义和易用的 DSL。Dataset 的 high-level API 支持大部分计算，例如 agg、select、sum、avg、map、filter 或者 groupBy 等计算。

（2）编译时就可捕捉语法错误，如图 6-5 所示。

	SQL	DataFrames	Datasets
Syntax Errors	Runtime	Compile Time	Compile Time
Analysis Errors	Runtime	Runtime	Compile Time

图 6-5 DataFrame、Dataset 编译时捕获错误信息

（3）使用 Catalyst 优化器：DataFrame 和 Dataset API 是建立在 Spark SQL 引擎之上的，它会使用 Catalyst 优化器来生成优化过的逻辑计划和物理查询计划。Scala、Java、Python 或 R 的 DataFrame/Dataset API 使得查询都进行相同的代码优化，以及空间和速度的效率提升。

（4）使用 Tungsten 进行内存管理：Spark 作为编译器可以理解 Dataset 类型的 JVM object，它能映射特定类型的 JVM object 到 Tungsten 内存管理，使用 Encoder。Tungsten 的 Encoder 可以有效地序列化 / 反序列化 JVM object，生成字节码来提高执行速度。RDD 与 Dataset 使用 cache 时，内存的大致统计信息如图 6-6 所示。

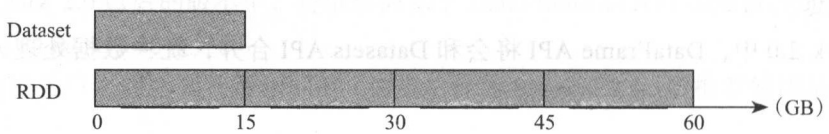


图 6-6 RDD、Dataset 使用 cache 时，内存使用情况。

Dataset/DataFrame 不足如下：

（1）DataFrame 不是类型安全的；

（2）API 不是面向对象风格的。

Dataset/DataFrame 使用场景如下：

（1）使用丰富的语义、high-level 抽象和特定领域语言 API；

（2）处理半结构化数据集需要 high-level 表达，如 filter、map、aggregation、average、sum、SQL 查询等，列式访问和使用 lambda 函数；

（3）编译时利用高度的 type-safety；

（4）使用统一和简化 API 及跨 Spark 的 Library；

（5）Python、R 熟悉者。

6.5.3 相互转换

RDD、DataFrame 和 Dataset 间可以互相转换。

（1）RDD->DataFrame：把 RDD 转换为 DataFrame 时，需要确定其 Schema，确定 RDD 的 Schema 有以下几种方法：在利用反射机制推断 RDD 模式时，需要首先定义一个 case class，因为，只有 case class 才能被 Spark 隐式地转换为 DataFrame；当无法提前定义 case class 时，就需要采用编程方式定义 RDD 模式。

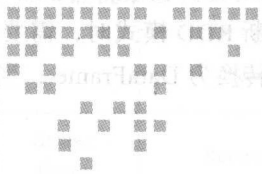
（2）RDD->Dataset

val ds = rdd.toDS()

（3）DataFrame/Dataset->RDD：把 DataFrame 或者 Dataset 转化成一个 RDD，只需简单的调用 .rdd 即可。

6.6　小结

本章主要介绍了 Spark MLlib 的一些内容，包括 MLlib 的生态、架构等内容，同时介绍了 Spark MLlib 算法底层依赖的基础内容，如数据类型、基础统计等，最后简单介绍了 RDD、DataFrame 与 Dataset 间的异同等。后续章节我们将通过一些实例，说明如何把前几章介绍的方法应用到具体实例中。



Chapter 7 第 7 章

构建 Spark ML 推荐模型

前面我们介绍了机器学习的一般步骤、如何探索数据、如何预处理数据、如何利用 Spark Ml 中的一些算法或 API，以及如何有效处理机器学习过程中的特征转换、特征选择、训练模型，并把这些过程流程化等。从本章开始，我们将通过实例，进一步阐述这些问题，并通过实例把相关内容有机结合起来。

本章主要介绍 Spark 机器学习中的协同过滤（Collaborative Filtering，CF）模型。协调过滤简单来说就是利用某个兴趣相投、拥有共同经验之群体的喜好来为使用户推荐其感兴趣的资讯，个人通过合作的机制给予资讯相当程度的回应（如评分）并记录下来以达到过滤的目的，进而帮助别人筛选资讯。回应不一定局限于特别感兴趣的，特别不感兴趣资讯的记录也相当重要。在日常生活中，人们实际上经常使用这种方法，如你哪天突然想看个电影，但你不知道具体看哪部，你会怎么做？大部分人会问周围的朋友，而我们一般更倾向于从兴趣或观点相近的朋友那里得到推荐。这就是协同过滤的思想。换句话说，就是借鉴和你相关的人群的观点来进行推荐。

本章介绍 Spark 的推荐模型，将按以下步骤进行：

- ❑ 推荐模型简介
- ❑ 加载数据到 HDFS
- ❑ Spark 读取数据
- ❑ 对数据进行探索
- ❑ 训练模型
- ❑ 组装任务
- ❑ 评估、优化模型

7.1　推荐模型简介

协同过滤常被用于推荐系统。这类技术目标在于填充“用户 – 商品”联系矩阵中的缺失项。Spark.ml 目前支持基于模型的协同过滤，其中用户和商品以少量的潜在因子来描述，用以预测缺失项。Spark.ml 使用交替最小二乘（ALS）算法来学习这些潜在因子。

ALS（Alternating Least Squares，交替最小二乘）特指使用交替最小二乘求解的一个协同推荐算法。它通过观察到的所有用户给产品的打分，来推断每个用户的喜好并向用户推荐适合的产品。我们看看表 7-1 所示的这个用户给物品打分的矩阵。

表 7-1　打分矩阵

用户 \ 商品	I1	I2	I3	…	Im
U1	2	3	?	…	3
U2	1	?	5	…	?
U3	?	?	3	…	4
…	…	…	…	…	3
U*n*	3	5	?	…	4

推荐系统的目标就是预测出符号“?”对应位置的分值（或把符号为“?”的项补全）。推荐系统基于这样一个假设：用户对物品的打分越高，表明用户越喜欢。因此，预测出用户对未评分物品的评分后，根据分值大小排序，把分值高（或前几名）的物品推荐给用户。具体如何实现？这里采用一种矩阵分解的方法，如图 7-1 所示。

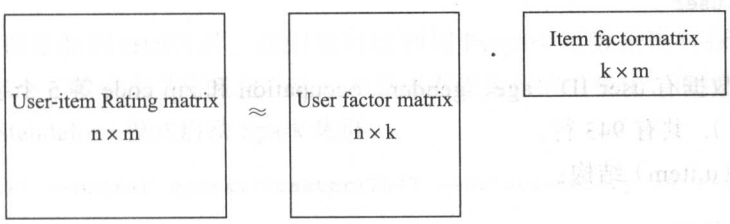


图 7-1　矩阵分解

ALS 的核心就是下面这个假设：被打分矩阵是近似低秩的。换句话说，一个 $n \times m$ 的打分矩阵 A 可以用两个小矩阵 $U(n \times k)$ 和 $V(k \times m)$ 的乘积来近似：

$$A \approx UV^T, k \ll m, n$$

这样我们就把整个系统的自由度从 O(nm) 一下降到了 O((n+m)k)。

在具体计算过程中，为了防止过度拟合，加上正则化参数：

$$L(U,V) = \sum_y (R_y - U_i V_J^T)^2 + \lambda(\|U_i\|_y^2 + \|V_i\|_y^2)$$

通过学习，我们得到一个代表用户的特征的低维（$n \times k$）用户特征因子，另一个代表物品特征的低维（$k \times m$）物品特征因子。特征因子的每一个维度代表一个隐性因子，比如对电

影来说，这些隐性因子可能是导演、演员等。当然，这些隐性因子是机器学习得到的，具体是什么含义我们不确定。

得到用户特征因子（$n \times k$）、物品特征因子（$k \times m$）后，计算用户相似度或物品相似度就可以转换为求两用户特征因子向量之间的相似度，或两物品特征因子向量之间的相似度。

矩阵分解 Spark 采用 ALS。它功能强大、效果理想而且易实现并行化，故特别适合 Spark 平台。

ALS 的实现原理是迭代式求解一系列最小二乘回归问题。每一次迭代时，固定用户因子或物品因子矩阵之一，然后用固定的这个及其评级数据更新另一个矩阵，循环迭代，直到模型收敛或迭代到指定次数。

7.2 数据加载

这里使用 MovieLens 100k 数据集，主要包括用户属性数据（u.user）、电影数据（u.item）、用户对电影的评级数据（u.data）及题材数据（u.genre）等。在把数据复制到 HDFS 之前，我们先大致了解一下相关数据。

用户数据（u.user）结构：

```
$ head -3 u.user
1|24|M|technician|85711
2|53|F|other|94043
3|23|M|writer|32067
$ wc -l u.user
943 u.user
```

可以看出用户数据有 user ID、age、gender、occupation 和 zip code 等 5 个字段，字段间隔符为竖线（“|”），共有 943 行。

电影数据（u.item）结构：

```
head -3 u.item
1|Toy Story
(1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Toy%20Story%20
(1995)|0|0|0|1|1|1|0|0|0|0|0|0|0|0|0|0|0|0|0
2|GoldenEye
(1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?GoldenEye%20
(1995)|0|1|1|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0
3|Four Rooms
(1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Four%20Rooms%20
(1995)|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0
$ wc -l u.item
1682 u.item
```

可以看出用户数据有 movie ID、title、release date 及其他属性，字段间隔符为竖线（“|”），共有 1682 行。

用户对电影评级数据（u.data）结构：

```
$ head -3 u.data
196     242     3       881250949
186     302     3       891717742
22      377     1       878887116
$ wc -l u.data
100000 u.data
```

可以看出用户数据有 user ID、movie ID、rating（1-5）和 timestamp 等 4 个字段，字段间隔符为制表符（“\t”），共有 100000 行。

电影题材数据（u.genre）：

```
$ head -3 u.genre
unknown|0
Action|1
Adventure|2
$wc -l u.genre
20 u.genre
```

这个数据只有题材及代码两个字段，以竖线分隔。共有 20 种电影题材。

把用户数据（u.user）复制到 HDFS 上，其他数据方法一样。

```
$ hadoop fs -put u.user /home/hadoop/data/
```

查看数据复制是否成功。

```
$ hadoop fs -ls /home/hadoop/data/
```

把相关数据复制到 HDFS 后，我们就可以利用 Pyspark 对数据进行探索或简单分析，这里使用 Pyspark 主要考虑其可视化功能，如果不需要数据的可视化，使用 Spark 即可。

以 Spark Standalone 模式启动 Spark 集群：

```
spark-shell --master spark://master:7077 --driver-memory 1G --total-executor-cores 4
```

导入需要的包或库：

```
import org.apache.spark.ml.evaluation.RegressionEvaluator
import org.apache.spark.ml.recommendation.ALS
import org.apache.spark.sql.SparkSession
import org.apache.spark.ml.Pipeline
```

因读入数据默认都是字符格式，故需要对数据进行格式转换。

```
// 定义个类，来保存一次评分
case class Rating(userId: Int, movieId: Int, rating: Float, timestamp: Long)
// 把一行转换成一个评分类
def parseRating(str: String): Rating = {
    val fields = str.split("\t")
```

```
        assert(fields.size == 4)
         Rating(fields(0).toInt, fields(1).toInt, fields(2).toFloat, fields(3).
toLong)
    }
```

读取并缓存数据，因后面需要多次使用这份数据。

```
val ratings = spark.read.textFile("hdfs://master:9000/home/hadoop/data/u.data")
    .map(parseRating)
    .cache()
```

7.3 数据探索

数据加载到 HDFS 后，我们便可对数据进行探索和分析，对用户数据的探索，大家可参考 2.4.3 节的相关内容。用户对电影评级的数据比较简单，这里我们简单查看一下导入数据抽样及统计信息。

抽样数据：

```
ratings.show(4)
+------+-------+------+---------+
|userId|movieId|rating|timestamp|
+------+-------+------+---------+
|   196|    242|   3.0|881250949|
|   186|    302|   3.0|891717742|
|    22|    377|   1.0|878887116|
|   244|     51|   2.0|880606923|
+------+-------+------+---------+
```

用户 ID、电影 ID、评级数据统计信息：

```
ratings.describe("userId","movieId","rating").show()
+-------+------------------+------------------+------------------+
|summary|            userId|           movieId|            rating|
+-------+------------------+------------------+------------------+
|  count|            100000|            100000|            100000|
|   mean|         462.48475|         425.53013|           3.52986|
| stddev|266.61442012750905|330.79835632558473|1.1256735991443214|
|    min|                 1|                 1|               1.0|
|    max|               943|              1682|               5.0|
+-------+------------------+------------------+------------------+
```

由此可知，该数据集共有 100000 条，评级最低为 1.0，最高为 5.0，平均 3.5 左右。

7.4 训练模型

这里数据比较简单，无须做数据转换和清理等数据预处理工作。在训练模型前，我们

需要把数据划分为几个部分，这里先随机划分成两部分，其中 80% 作为训练集，20% 作为测试集。后续我们在性能优化时将采用另一种划分方式，然后，比较使用不同划分方法对模型性能或泛化能力的影响。

```
val Array(training, test) = ratings.randomSplit(Array(0.8, 0.2),seed=1234)

val als = new ALS()
    .setMaxIter(10)
    .setRank(10)
    .setRegParam(0.01)
    .setNonnegative(true)
    .setUserCol("userId")
    .setItemCol("movieId")
    .setRatingCol("rating")
```

ALS 参数说明：

- numBlocks 是用于并行化计算的分块个数 (默认为 10);
- rank 是模型中隐语义因子的个数（默认为 10);
- maxIter 是迭代的次数（默认为 10);
- regParam 是 ALS 的正则化参数（默认为 1.0);
- implicitPrefs 该参数适用显性反馈，还是适用隐性反馈（默认是 false，即用显性反馈);
- alpha 该参数决定了偏好行为强度的基准（默认为 1.0);
- nonnegative 对最小二乘法使用非负的限制（默认为 false);
- UserCol、ItemCol、RatingCol 为输入字段。

7.5　组装

1）创建流水线，把数据转换、模型训练等任务组装在一条流水线上。

```
val pipeline = new Pipeline().setStages(Array(als))
```

2）训练模型：

```
val model = pipeline.fit(training)
```

3）作出预测：

```
val predictions = model.transform(test)
```

4）查看预测值与原来的值：

```
predictions.show(5)
+------+-------+------+----------+----------+
```

```
|userId|movieId|rating|timestamp|prediction|
+------+-------+------+---------+----------+
|   222|    148|   2.0|881061164| 3.1357265|
|   330|    148|   4.0|876544781| 3.9583592|
|   224|    148|   3.0|888104154| 3.9787998|
|   618|    148|   3.0|891309670| 2.7060091|
|   896|    148|   2.0|887160606| 2.8391676|
+------+-------+------+---------+----------+
```

7.6 评估模型

1）预测时会产生 NaN，即 NaN 表示不推荐（预测时产生 NaN 是 spark2.1 ALS 中的一个 bug，该 bug 在 2.2 版本中将修复）。

```
predictions.filter(predictions("prediction").isNaN).select("userId","movieId","rating","prediction").count()
```

2）删除含 NaN 的值的行，NaN 有一定合理性，不推荐，但为评估指标，可以先过滤这些数。

```
val predictions1= predictions.na.drop()

val evaluator = new RegressionEvaluator()
    .setMetricName("rmse")
    .setLabelCol("rating")
    .setPredictionCol("prediction")
val rmse = evaluator.evaluate(predictions1)
```

3）运行结果为：

```
rmse: Double = 1.016902715345917
```

7.7 模型优化

模型训练前，对数据集随机划分训练集和测试集，训练模型时选择了一些参数值，每种参数只选择一个值。这些参数的组合是否是合理或最优的？除了随机划分数据集，是否还有更好的划分方法？以下我们通过一个实例进行说明。

因预测值比较特殊，故这里我们选择自定义模型优化方法。首先，我们采用交叉验证方法，把数据集分成三部分，即训练集、验证集、测试集，对每种参数选择两种或两种以上，然后，对模型进行训练，最后利用自定义模型评估函数，在多种方案中，应先选择一种最优方案或最优模型。

```
// 导入一些包
```

```
import org.apache.spark.sql.Row
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.DataFrame
import org.apache.spark.ml.recommendation.{ALS,ALSModel}

//将样本评分表分成 3 个部分，分别用于训练 (60%)、校验 (20%) 和测试 (20%)
val splits = ratings.randomSplit(Array(0.6, 0.2,0.2),12)

//把训练样本缓存起来，加快运算速度
val training = splits(0).cache()
val validation = splits(1).toDF.cache()
val test = splits(1).toDF.cache()
//计算各集合总数
val numTraining = training.count()
val numValidation = validation.count()
val numTest = test.count()

//训练不同参数下的模型，并在校验集中验证，获取最佳参数下的模型
val ranks = List(10, 20)
val lambdas = List(0.01, 0.1)
val numIters = List(5, 10)
var bestModel: Option[ALSModel] = None
var bestValidationRmse = Double.MaxValue
var bestRank = 0
var bestLambda = 1.0
var bestNumIter = 1

def computeRmse(model:ALSModel,data:DataFrame,n:Long):Double = {
    val predictions = model.transform(data)
      val p1=predictions.na.drop().rdd.map{ x =>((x(0),x(1)),x(2))}.
join(predictions.rdd.map{ x =>((x(0),x(1)),x(4))}).values
    math.sqrt(p1.map( x => (x._1.toString.toDouble - x._2.toString.toDouble) * (x._1.
toString.toDouble - x._2.toString.toDouble)).reduce(_+_)/n)
    }

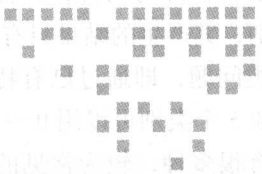
for (rank <- ranks; lambda <- lambdas; numIter <- numIters) {
val als = new ALS()
    .setMaxIter(numIter)
    .setRegParam(lambda)
    .setRank(rank)
.setNonnegative(true)
    .setUserCol("userId")
    .setItemCol("movieId")
    .setRatingCol("rating")
val model = als.fit(training)
val validationRmse = computeRmse(model, validation, numValidation)
    println("RMSE(validation) = " + validationRmse + " for the model trained with
rank = "
    + rank + ",lambda = " + lambda + ",and numIter = " + numIter + ".")
      if (validationRmse < bestValidationRmse) {
```

```
            bestModel = Some(model)
            bestValidationRmse = validationRmse
            bestRank = rank
            bestLambda = lambda
            bestNumIter = numIter
        }
    }
}
// 运行结果
RMSE(validation) = 1.0664516122491705 for the model trained with rank = 10,lambda
                 = 0.01,and numIter = 5.
RMSE(validation) = 1.0773258157269512 for the model trained with rank = 10,lambda
                 = 0.01,and numIter = 10.
RMSE(validation) = 0.9509095542582248 for the model trained with rank = 10,lambda
                 = 0.1,and numIter = 5.
RMSE(validation) = 0.9390664107785451 for the model trained with rank = 10,lambda
                 = 0.1,and numIter = 10.
RMSE(validation) = 1.1024492428290906 for the model trained with rank = 20,lambda
                 = 0.01,and numIter = 5.
RMSE(validation) = 1.1242105743040174 for the model trained with rank = 20,lambda
                 = 0.01,and numIter = 10.
RMSE(validation) = 0.9393089637028184 for the model trained with rank = 20,lambda
                 = 0.1,and numIter = 5.
RMSE(validation) = 0.9383240505365207 for the model trained with rank = 20,lambda
                 = 0.1,and numIter = 10.

// 用最佳模型预测测试集的评分，并计算和实际评分之间的均方根误差 (RMSE)
val testRmse = computeRmse(bestModel.get, test, numTest)
testRmse: Double = 0.9383240505365207
// 比优化前的 rmse: Double = 1.016902715345917 提高了 7.6% 左右。
// 打印最优模型中的各参数值
println("The best model was trained with rank = " + bestRank + " and lambda = "
                                                  + bestLambda
    + ", and numIter = " + bestNumIter + ", and its RMSE on the test set is " +
                        testRmse + ".")
// 最佳模型相关参数
The best model was trained with rank = 20 and lambda = 0.1, and numIter = 10, and
                                        its RMSE on the test set is 0.9383240505365207.
```

7.8 小结

本章介绍了推荐模型的一般方法，Spark 推荐模型的原理和算法等，然后通过一个实例具体说明了实施 Spark 推荐模型的一般步骤、使用自定义函数优化模型等内容。下一章将以 Spark ML 的分类模型为例，进一步说明如何使用 Spark ML 提供的特征选取、特征转换、流水线、交叉验证等函数或方法。



第 8 章 Chapter 8

构建 Spark ML 分类模型

在上一章中，我们通过实例介绍了 Spark 中基于协同过滤的推荐模型，了解了推荐模型的原理以及使用场景、使用流水线组装任务、使用自定义函数优化模型等。这一章我们将就 Spark 中的分类模型为例，进一步说明如何使用 Spark ML 中特征选取、特征转换、流水线、模型选择或优化等方法，简化、规范化、流程化整个机器学习过程。

分类、回归和聚类是机器学习中重要的几个分支，也是日常数据处理与分析中最常用的手段。这几类算法有着较高的成熟度，原理也较容易理解，且有着不错的效果，深受数据分析师们的喜爱。本章以 Spark ML 分类模型为例，主要包括以下内容：

- ❑ 简单介绍几种分类模型
- ❑ 加载数据
- ❑ 探索加载后的数据
- ❑ 预处理数据
- ❑ 把各种任务组装到流水线上
- ❑ 模型调优

8.1 分类模型简介

分类通常是指将事物分成不同的类别，属于监督学习中的一种形式，通过大量的样本数据构成一个训练集，每个样本有多个属性，属性可以是连续型，也可以是离散型。其中每一个属性我们都称为事物的类别属性，用来标识该样本所属的类别。简而言之，分类就是通过一组代表物体、事件等的相关属性来判断其类别。

分类问题通常会分为两种，一种是最简单的二分类问题，即通过已有特征属性判断事物或事件的类别，其产生的结果只有 0 和 1，即要么属于该类别，要么不属于该类别。另一种是多类别分类问题，即通过已有特征属性判断事物或事件的类别，其产生的结果可能有多个类别，比如 5 个类别可以用 0 ～ 4 来标识其类别。

分类模型有很多种，较为常见的几种模型分别为：线性模型、决策树模型、朴素贝叶斯模型、BP 神经网络模型。本章就线性模型、决策树模型以及朴素贝叶斯模型的使用进行详细介绍。

1）线性模型：模型简单，且容易扩展到较大的数据集；

2）决策树模型：具有强大的非线性技术，不要求样本数据标准化，易扩展到多分类问题，在很多情况下能够获得不错的性能；

3）朴素贝叶斯：模型简单、易训练，并且具有高效和并行的优点。

值得注意的是，Spark ML 目前支持的分类算法有：基于线性模型、决策树和朴素贝叶斯的二分类模型，以及基于决策树和朴素贝叶斯的多类别分类模型。对于线性模型中支持向量机（SVM）的模型目前还只能在 MLlib 使用，且不适用于流水线和交叉验证的方法，这里将不作探讨。

分类模型在日常生活中，有着很多的实际应用，以下是一些常见例子：

1）对图片、视频或者声音的分类；

2）对新闻、网页或者其他内容标记类别；

3）发现垃圾邮件、垃圾页面、网络入侵和其他恶意行为；

4）银行对贷款风险的评估，预测是否会拖欠贷款；

5）预测用户是否会购买某个产品或服务；

6）信用卡分级，将信用卡申请者分为低、中、高风险；

7）基于运营商数据的个人征信评估；

8）预测市民出行是否选乘公交；

9）预测互联网用户对在线广告的点击概率；

10）中文句子类别精准分析；

11）国家电网客户用电异常行为分析；

12）客户流失率预测。

8.1.1 线性模型

线性模型主要通过对样本的预测结果（通常称为目标或者因变量）进行建模，即通过样本数据进行训练得到线性预测函数，从而应用该函数对测试数据的输入变量（特征或者自变量）进行结果的预测。

逻辑回归是线性分类模型中使用最广泛的一个，它是一个概率性模型，即预测的结果只能是 0 ～ 1。逻辑回归其实只是在线性回归的基础上，套用了一个逻辑函数（如 Sigmoid

函数)。对于二分类来说，逻辑回归的输出等价于模型预测某个数据点属于正类的概率估计。

8.1.2　决策树模型

决策树模型是一个强大的非概率模型，可以用来表示复杂的非线性模式和特征的相互关系。决策树是一个树状结构（可以是二叉树或非二叉树），由节点和有向边组成。节点通常有两种类型：内部节点和叶子节点。内部节点表示事物的一个特征属性，叶子节点则表示一个类。每个非叶子节点表示一个特征属性上的测试，每个分支表示这个特征属性在某个值域上的输出，而叶子节点存放的是一个类别。

特征选择的目标使得分类后的数据集比较纯，如何度量纯度？可以通过数据集划分前后的信息变化来确定，或信息增益来确定。信息增益（Info Gain）最大的特征就是最好选择，如何计算信息增益？集合信息的度量方式称为熵（Entropy）。

假设变量 $X=\{x_1, x_2,..., x_i,..., x_n\}$，其中每个元素对应的概率（比例）为 $P=\{p_1, p_2, ..., p_i, ..., p_n\}$，则对应熵的计算公式：$E(X)=-\sum_{i=0}^{n} p_i \log_2(p_i)$。而信息增益（Info-Gain）指的就是熵的减少量：$IGain(S, A)=E(S)-E(A)$。公式中 S 和 A 分别代表数据划分前后的数据集。

决策树是一个贪心算法递归地将特征空间划分为两个部分，在同一个叶子节点的数据最后会拥有同样的标签。每次划分通过贪心算法以获得最大信息增益为目的，从可选择的分裂方式中选择最佳的分裂节点。节点不纯度由节点所含类别的同质性来衡量。为分类提供两种不纯度衡量（基尼不纯度和熵），为回归提供一种不纯度衡量（方差）。

决策树的决策过程是从根节点开始，测试待分类项中的特征属性，并按照其值选择输出分支，直到到达叶子节点，将叶子节点存放的类别作为决策结果。我们以图 8-1 用户是否购买计算机为例，对决策树的决策过程进行说明。

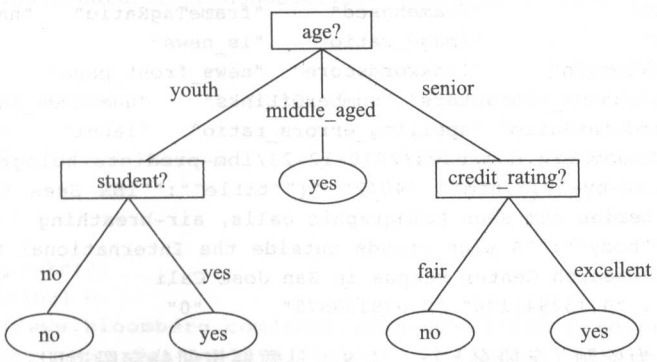


图 8-1　决策树

整个过程：首先从用户的年龄开始判断，如果用户中年，那么就可能会购买电脑；如果是年轻人，进行下一步判断，如果是学生就可能会购买，否则不会购买。如果是年长者，

那么根据信用评分来进行下一步的判断：如果是一般的，那么不会购买；如果是极好的，那么就可能购买。整个流程从根节点开始不断向下做判断，呈现树状的结构。关于根节点以及叶子节点的选择是通过信息熵进行计算而做出选择，这些 Spark ML 已经帮我们做了，我们只要关心如何构建模型即可。

除了信息增益为衡量指标外，还有信息增益率，基尼指数（Gini）等。其中信息增益（Info Gain）用于 ID3，Gini 用于 CART，信息增益率（Info Gain Ratio）用于 C4.5。

8.1.3 朴素贝叶斯模型

朴素贝叶斯分类是一个简单概率模型，通过计算给定的带分类项在每个类别的概率来进行预测，哪个概率最大就认为待分类项属于哪个类别。

关于朴素贝叶斯详细的原理，在维基百科中有更为详细的数学公式解释：http://en.wikipedia.org/wiki/Naive_Bayes_classifier。

8.2 数据加载

这里的数据选择为某比赛的数据集，用来预测推荐的一些页面是短暂（昙花一现）还是长久（长时流行）。原数据集为 train.tsv，存放路径在 /home/hadoop/data/train.tsv。

先使用 shell 命令对数据进行试探性的查看，并做一些简单的数据处理。

1）查看前两行数据：

```
$ head -2 train.tsv

"url"  "urlid"  "boilerplate"     "alchemy_category"  "alchemy_category_score"
  "avglinksize"          "commonlinkratio_1"       "commonlinkratio_2"
  "commonlinkratio_3"    "commonlinkratio_4"       "compression_ratio"
  "embed_ratio"          "framebased"     "frameTagRatio"    "hasDomainLink"
  "html_ratio"           "image_ratio"    "is_news"
  "lengthyLinkDomain"    "linkwordscore"  "news_front_page"
"non_markup_alphanum_characters" "numberOfLinks"    "numwords_in_url"
  "parametrizedLinkRatio" "spelling_errors_ratio"   "label"
"http://www.bloomberg.com/news/2010-12-23/ibm-predicts-holographic-calls-air-
breathing-batteries-by-2015.html" "4042" "{""title"":""IBM Sees Holographic Calls
Air Breathing Batteries ibm sees holographic calls, air-breathing
batteries"",""body"":""A sign stands outside the International Business Machines
Corp IBM Almaden Research Center campus in San Jose Cali          "8"
.............. "0.152941176" "0.079129575"      "0"
```

数据集中的第 1 行为标题（字段名）行，表 8-1 是数据集部分字段说明。

紧接着的 22 列为各种数值和页面的所属类别。

最后一列为标签，用来标识网页是长久的还是暂时的。取值有 0 和 1，1 标识此页面是长久的，0 表示此页面是暂时的。

2）查看文件记录总数：

```
$ cat train.tsv |wc -l
7396
```

结果显示：数据集一共有 7396 条数据

3）由于 textFile 目前不好过滤标题行数据，为便于 spark 操作数据，需要先删除标题。

```
$ sed  1d train.tsv >train_noheader.tsv
```

表 8-1　数据集部分字段说明

字段	说明
url	页面的 url
urlid	页面 ID
boilerplate	页面的文本内容
alchemy_category	页面所属类别
label	页面类别

4）将数据文件上传到 hdfs：

```
$ hdfs dfs -put train_noheader.tsv /data
```

5）查看是否上成功：

```
hadoop@master:~/data$ hdfs dfs -ls /data
17/05/24 00:46:20 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Found 1 items
-rw-r--r--    1 hadoop supergroup    21972457 2017-05-24 00:46
/data/train_noheader.tsv
```

6）启动 Spark Shell：

```
spark-shell --master spark://master:7077 --driver-memory 4G --total-executor-
cores 4
```

7）通过 sc 对象的 textFile 方法，由本地文件数据创建 RDD：

```
scala> val rawData=sc.textFile("hdfs://master:9000/data/train_noheader.tsv")
rawData: org.apache.spark.rdd.RDD[String] =
hdfs://master:9000/data/train_noheader.tsv MapPartitionsRDD[1] at textFile at
<console>:24
```

8.3　数据探索

1）查看数据前两行：

```
scala> rawData.take(2)
res0: Array[String] =
Array("http://www.bloomberg.com/news/2010-12-23/ibm-predicts-holographic-
calls-air-breathing-batteries-by-2015.html"    "4042"    "{""title"":""IBM Sees
Holographic Calls Air Breathing Batteries ibm sees holographic calls, air-breathing
batteries"",""body"":""A sign stands outside the International Business Machines Corp
IBM Almaden Research Center campus in San Jose California Photographer Tony Avelar
Bloomberg Buildings stand at the International Business Machines Corp IBM Almaden
Research Center campus in the Santa Teresa Hills of San Jose California Photographer
```

```
Tony Avelar Bloomberg By 2015 your mobile phone will project a 3 D image of anyone
who calls and your laptop will be powered by kinetic energy At least that s what
International Business Machines Corp sees in...
```

由上面可以看到，得到的是只有一行字符串数组。通过常看源文件，我们可以发现字段间由制表符 (\t) 分隔。由于后续的算法我们不需要时间戳以及网页的内容，所以这里先将其过滤掉。下面我们获取每个属性。

2）根据以上分析，对数据进行处理，并生成新的 RDD：

```
scala> val records = rawData.map(line => line.split("\t"))
records: org.apache.spark.rdd.RDD[Array[String]] = MapPartitionsRDD[2] at map at
<console>:26
```

3）查看数据结构：

```
scala> records.first
res4: Array[String] =
Array("http://www.bloomberg.com/news/2010-12-23/ibm-predicts-holographic-
calls-air-breathing-batteries-by-2015.html", "4042", "{""title"":""IBM Sees
Holographic Calls Air Breathing Batteries ibm sees holographic calls, air-breathing
batteries"",""body"":""A sign stands outside the International Business Machines Corp
IBM Almaden Research Center campus in San Jose California Photographer Tony Avelar
Bloomberg Buildings stand at the International Business Machines Corp IBM Almaden
Research Center campus in the Santa Teresa Hills of San Jose California Photographer
Tony Avelar Bloomberg By 2015 your mobile phone will project a 3 D image of anyone
who calls and your laptop will be powered by kinetic energy At least that s what
International Business Machines Corp sees ...
```

4）查看总的数据行数：

```
scala> records.count
res5: Long = 7395
```

5）查看每一行数据的列数：

```
scala> records.first.size
res6: Int = 27
```

6）获取第一行的某个值：

```
scala> records.first.take(2)
res22: Array[String] =
Array("http://www.bloomberg.com/news/2010-12-23/ibm-predicts-holographic-calls-
air-breathing-batteries-by-2015.html", "4042")
```

8.4 数据预处理

在上述的基础上，我们需要对数据进行进一步的数据清洗，具体内容如下：

- 去掉引号。
- 把标签列（即最后一列）转换为整数。
- 把第 4 列中的？转换为 0.0。
- 使用 LabeledPoint 方法给数据打标签，即：把标签及特征转换为 LabeledPoint 实例（一种分类算法的需求格式），同时把特征向量存储到 ML 的 Vector 中。



注意　LabeledPoint 格式数据如：label index1:value1 index2:value2...，可通过 MLUtils.loadLibSVMFile（sc,“path”）直接读入。

1）导入 LabeledPoint：

```
scala> import org.apache.spark.ml.feature.LabeledPoint
```

2）导入 Vectors 矢量方法：

```
scala> import org.apache.spark.ml.linalg.Vectors
```

3）对数据进行 1 ～ 4 步的数据清洗工作：

```
val data = records.map { r =>
      val trimmed = r.map(_.replaceAll("\"", ""))
      val label = trimmed(r.size - 1).toInt
       val features = trimmed.slice(4, r.size - 1).map(d => if (d == "?") 0.0
else d.toDouble)
      LabeledPoint(label, Vectors.dense(features))
}
```

上述代码可通过复制粘贴到代码行中，使用 :paste，粘贴过后按下 Ctrl+D 即可。

4）考虑到使用朴素贝叶斯算法时，数据需不小于 0，故需要做些处理。

```
val nbData = records.map { r =>
        val trimmed = r.map(_.replaceAll("\"", ""))
        val label = trimmed(r.size - 1).toInt
        val features = trimmed.slice(4, r.size - 1).map(d => if (d == "?") 0.0 else
d.toDouble).map(d => if (d < 0) 0.0 else d)
        LabeledPoint(label, Vectors.dense(features))
}
```

5）查看清理后数据集的前两行数据：

```
scala> data.take(2)
res0: Array[org.apache.spark.ml.feature.LabeledPoint] =
Array((0.0,[0.789131,2.055555556,0.676470588,0.205882353,0.047058824,
0.023529412,0.443783175,0.0,0.0,0.09077381,0.0,0.245831182,0.003883495,1.0,1.0,24.0,
0.0,5424.0,170.0,8.0,0.152941176,0.079129575]),
(1.0,[0.574147,3.677966102,0.50802139,0.288770053,0.213903743,0.144385027,0.4686
48998,0.0,0.0,0.098707403,0.0,0.203489628,0.088652482,1.0,1.0,40.0,0.0,4973.0,187.0,
9.0,0.181818182,0.125448029]))
```

6）通过 RDD 创建 DataFrame：

```
scala> val df = spark.createDataFrame(data)
df: org.apache.spark.sql.DataFrame = [label: double, features: vector]

scala> val nbDF = spark.createDataFrame(nbData)
nbDF: org.apache.spark.sql.DataFrame = [label: double, features: vector]
```

7）查看 df 和 nbDF 的数据：

```
scala> df.show(10)    // 查看 df 的前 10 行数据
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|[0.789131,2.05555...|
|  1.0|[0.574147,3.67796...|
|  1.0|[0.996526,2.38288...|
|  1.0|[0.801248,1.54310...|
|  0.0|[0.719157,2.67647...|
|  0.0|[0.0,119.0,0.7454...|
|  1.0|[0.22111,0.773809...|
|  0.0|[0.0,1.883333333,...|
|  1.0|[0.0,0.471502591,...|
|  1.0|[0.0,2.41011236,0...|
+-----+--------------------+
only showing top 10 rows

// 查看 nbDF 的第一行数据，或者使用 nbDF.first 也是一样的
scala> nbDF.head
res21: org.apache.spark.sql.Row =
[0.0,[0.789131,2.055555556,0.676470588,0.205882353,0.047058824,0.023529412,0.443
783175,0.0,0.0,0.09077381,0.0,0.245831182,0.003883495,1.0,1.0,24.0,0.0,5424.0,170.0,
8.0,0.152941176,0.079129575]]
```

8）查看 df 和 nbDF 的 Schema 的信息和数据总行数：

```
scala> df.printSchema
root
 |-- label: double (nullable = true)
 |-- features: vector (nullable = true)

scala> df.count
res4: Long = 7395

scala> nbDF.printSchema
root
 |-- label: double (nullable = true)
 |-- features: vector (nullable = true)

scala> nbDF.count
res24: Long = 7395
```

9）随机地将数据进行划分，80% 用于训练集，20% 用于测试集：

```
scala>  val Array(trainingData, testData) = df.randomSplit(Array(0.8, 0.2),seed
                                          = 1234L)
trainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =[label:
double, features: vector]
testData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [label:
double, features: vector]

scala> val Array(nbTrainingData, nbTestData) = nbDF.randomSplit(Array(0.8, 0.2),
                                               seed = 1234L)
nbTrainingData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] = [label:
double, features: vector]
nbTestData: org.apache.spark.sql.Dataset[org.apache.spark.sql.Row] =[label:
double, features: vector]
```

10）查看训练数据和测试数据的总行数：

```
scala> trainingData.count
res8: Long = 5912

scala> testData.count
res9: Long = 1483
```

11）由于后续使用网格参数和交叉验证的时候，需要多次使用到训练集和测试集，所以将这两者载入内存，可大大提高性能。

```
scala> trainingData.cache
res10: trainingData.type = MapPartitionsRDD[24] at randomSplit at <console>:35

scala> testData.cache
res11: testData.type = MapPartitionsRDD[25] at randomSplit at <console>:35

scala> nbTrainingData.cache
res27: nbTrainingData.type = [label: double, features: vector]

scala> nbTestData.cache
res28: nbTestData.type = [label: double, features: vector]
```

12）导入逻辑回归分类器、决策树模型以及朴素贝叶斯模型。

```
scala> import
org.apache.spark.ml.classification.{LogisticRegression,LogisticRegressionModel}

scala> import org.apache.spark.ml.classification.{NaiveBayes,NaiveBayesModel}

scala> import
org.apache.spark.ml.classification.{DecisionTreeClassifier,DecisionTreeClassificati
onModel}
```

13）创建贝叶斯模型，设置初始参数：

```
scala> val nb = new NaiveBayes().setLabelCol("label").setFeaturesCol("features")
nbModel: org.apache.spark.ml.classification.NaiveBayes = nb_050f7aa0718e
```

14）通过朴素贝叶斯训练模型，对数据进行预测：

```
// 训练数据
scala> val nbModel = nb.fit(nbTrainingData)
nbModel: org.apache.spark.ml.classification.NaiveBayesModel = NaiveBayesModel
(uid=nb_63013179fe1f) with 2 classes

// 预测数据
scala> val nbPrediction = nbModel.transform(nbTestData)
nbPrediction: org.apache.spark.sql.DataFrame = [label: double, features:
vector ... 3 more fields]

scala> nbPrediction.show(10)
+-----+--------------------+--------------------+--------------------+----------+
|label|            features|       rawPrediction|         probability|prediction|
+-----+--------------------+--------------------+--------------------+----------+
|  0.0|[0.0,0.0,0.0,0.0,...|[-46.748642409367...|[0.83980023102260...|       0.0|
|  0.0|[0.0,0.0,0.0,0.0,...|[-28.455469606463...|[0.80560314910869...|       0.0|
|  0.0|[0.0,0.0,0.0,0.0,...|[-21.419660085849...|[0.74277974559336...|       0.0|
|  0.0|[0.0,0.253731343,...|[-566.66641956697...|[1.0,1.7019208090...|       0.0|
|  0.0|[0.0,0.5,0.0,0.0,...|[-85.270246662200...|[0.83409619579026...|       0.0|
|  0.0|[0.0,0.5,0.0,0.0,...|[-109.88609079237...|[0.94215720717655...|       0.0|
|  0.0|[0.0,0.563636364,...|[-645.84504343631...|[3.97514834896843...|       1.0|
|  0.0|[0.0,0.590163934,...|[-2040.0838024687...|[0.99999724227148...|       0.0|
|  0.0|[0.0,0.677966102,...|[-432.36145227604...|[0.99888109472905...|       0.0|
|  0.0|[0.0,0.7,0.111111...|[-222.48044531991...|[0.99999999305131...|       0.0|
+-----+--------------------+--------------------+--------------------+----------+
only showing top 10 rows
```

15）朴素贝叶斯准确性统计：

```
// t1 存放预测值的数组，t2 存放测试数据标签值，t3 存放测试数据总行数
scala>val (t1, t2, t3) = (nbPrediction.select("prediction").collect,
nbTestData.select("label").collect,nbTestData.count.toInt)

// t4 为累加器
scala> var t4 = 0
t4: Int = 0

// 遍历循环，统计正确预测的次数
scala> for(i <- 0 to t3-1){if(t1(i)==t2(i)) t4+=1}

// 查看预测正确的个数
scala> t4
res63: Int = 840

// 计算准确率
scala> val nbAccuracy = 1.0*t4/t3
```

```
nbAccuracy: Double = 0.5664194200944033
```

可以看到，朴素贝叶斯的准确率为 56.6419%。

8.5　组装

1）导入特征索引类：

```
scala>import org.apache.spark.ml.feature.{ VectorIndexer, VectorIndexerModel}
```

2）建立特征索引：

```
scala> val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").fit(df)
featureIndexer: org.apache.spark.ml.feature.VectorIndexerModel =
vecIdx_b73ca1435eea
```

3）创建逻辑回归模型：

```
scala> val lr = new
LogisticRegression().setLabelCol("label").setFeaturesCol("indexedFeatures").
setMaxIter(10).setRegParam(0.001)
lrModel: org.apache.spark.ml.classification.LogisticRegression =
logreg_9bec21f2262f
```

4）创建决策树模型：

```
scala> val dt = new
DecisionTreeClassifier().setLabelCol("label").setFeaturesCol("indexedFeatures").
setImpurity("entropy").setMaxBins(100).setMaxDepth(5).setMinInfoGain(0.01)
dtModel: org.apache.spark.ml.classification.DecisionTreeClassifier =
dtc_8a3a01185f6b
```

部分参数说明如下：

- ❑ setImpurity(“entropy”)：指定信息熵，这里为 entropy。
- ❑ setMaxBins(100)：离散化“连续特征”的最大划分数，这里设置为 100。
- ❑ setMaxDepth(5)：树的最大深度，这里为 5。
- ❑ setMinInfoGain(0.01)：一个节点分裂的最小信息增益，值为 [0,1]。

5）导入网格参数和交叉验证：

```
scala> import org.apache.spark.ml.tuning.{ ParamGridBuilder, CrossValidator }
```

6）导入流水线：

```
import org.apache.spark.ml.{Pipeline,PipelineModel}
```

7）导入评估器：

```
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
```

8）配置两条流水线：一条是逻辑回归的流水线，包含两个 stages（featureIndexer 和 lr）；一条是决策树回归的流水线，包含两个 stages（featureIndexer 和 dt）。

```
scala> val lrPipeline = new Pipeline().setStages(Array(featureIndexer,lr))
lrPipeline: org.apache.spark.ml.Pipeline = pipeline_64c542dff42e
scala> val dtPipeline = new Pipeline().setStages(Array(featureIndexer,dt))
dtPipeline: org.apache.spark.ml.Pipeline = pipeline_b9ed2ccc2108
```

8.6 模型优化

1）分别配置网格参数，使用 ParamGridBuilder 构造一个 parameter grid：

```
scala> :paste
val lrParamGrid = new ParamGridBuilder()
  .addGrid(lr.regParam,Array(0.1,0.3,0.5))
  .addGrid(lr.maxIter, Array(10,20,30))
  .build()

scala> :paste
val dtParamGrid = new ParamGridBuilder()
  .addGrid(dt.maxDepth, Array(3,5,7))
  .build()
```

2）分别实例化交叉验证模型：

```
val evaluator = new BinaryClassificationEvaluator()

scala> :paste
val lrCV = new CrossValidator()
  .setEstimator(lrPipeline)
  .setEvaluator(evaluator)
  .setEstimatorParamMaps(lrParamGrid)
  .setNumFolds(2)
lrCV: org.apache.spark.ml.tuning.CrossValidator = cv_b25c7e0f1be7

scala> :paste
val dtCV = new CrossValidator()
  .setEstimator(dtPipeline)
  .setEvaluator(evaluator)
  .setEstimatorParamMaps(dtParamGrid)
  .setNumFolds(2)

dtCV: org.apache.spark.ml.tuning.CrossValidator = cv_5176e642601d
```

3）通过交叉验证模型，获取最优参数集，并测试模型：

```
scala>val lrCvModel = lrCV.fit(trainingData)
lrCvModel: org.apache.spark.ml.tuning.CrossValidatorModel = cv_b25c7e0f1be7
scala> val dtCvModel = dtCV.fit(trainingData)
dtCvModel: org.apache.spark.ml.tuning.CrossValidatorModel = cv_5176e642601d

scala> val lrPrediction = lrCvModel.transform(testData)
lrPrediction: org.apache.spark.sql.DataFrame = [label: double, features: vector
... 4 more fields]

scala> val dtPrediction = dtCvModel.transform(testData)
dtPrediction: org.apache.spark.sql.DataFrame = [label: double, features: vector
... 4 more fields]
```

4）查看数据：

```
scala> lrPrediction.select("label","prediction").show(10)
+-----+----------+
|label|prediction|
+-----+----------+
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       1.0|
|  0.0|       1.0|
|  0.0|       1.0|
|  0.0|       1.0|
|  0.0|       0.0|
|  0.0|       0.0|
+-----+----------+
only showing top 10 rows

scala> dtPrediction.select("label","prediction").show(10)
+-----+----------+
|label|prediction|
+-----+----------+
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       0.0|
|  0.0|       1.0|
|  0.0|       1.0|
|  0.0|       1.0|
|  0.0|       0.0|
+-----+----------+
only showing top 10 rows
```

5）查看逻辑回归匹配模型的参数：

```
scala> val lrBestModel = lrCvModel.bestModel.asInstanceOf[PipelineModel]
lrBestModel: org.apache.spark.ml.PipelineModel = pipeline_64c542dff42e
scala> val lrModel = lrBestModel.stages(1).asInstanceOf[LogisticRegressionModel]
lrModel: org.apache.spark.ml.classification.LogisticRegressionModel =
logreg_100994a23a48

scala> lrModel.getRegParam
res20: Double = 0.1

scala> lrModel.getMaxIter
res21: Int = 20
```

6）查看决策树匹配模型的参数：

```
scala> val dtBestModel = dtCvModel.bestModel.asInstanceOf[PipelineModel]
dtBestModel: org.apache.spark.ml.PipelineModel = pipeline_b9ed2ccc2108

scala> val dtModel =
dtBestModel.stages(1).asInstanceOf[DecisionTreeClassificationModel]
dtModel: org.apache.spark.ml.classification.DecisionTreeClassificationModel =
DecisionTreeClassificationModel (uid=dtc_b10fd9474309) of depth 4 with 17 nodes

scala> dtModel.getMaxDepth
res24: Int = 3

scala> dtModel.numFeatures
res25: Int = 22
```

7）统计逻辑回归的预测正确率：

```
// t_lr 为逻辑回归预测值的数组 ,t_dt 为决策树预测值的数组
// t_label 为测试集的标签值的数组
scala> val (t_lr, t_dt, t_label, t_count) =
(lrPrediction.select("prediction").collect,
dtPrediction.select("prediction").collect,testData.select("label").
collect,testData.count.toInt)

// c_lr 为统计逻辑回归预测正确个数的累加器
//c_dt 为统计决策树预测正确个数的累加器
scala> var Array(c_lr,c_dt) = Array(0,0)
t4: Int = 0

// 遍历循环，统计逻辑回归正确预测的次数
scala> for(i <- 0 to t_count-1){if(t_lr(i)==t_label(i)) c_lr+=1}

scala> c_lr
res5: Int = 899

// 统计逻辑回归正确率
scala> 1.0*c_lr/t_count
res6: Double = 0.6062036412677007
```

```
// 遍历循环，统计逻辑回归正确预测的次数
scala> for(i <- 0 to t_count-1){if(t_dt(i)==t_label(i)) c_dt+=1}

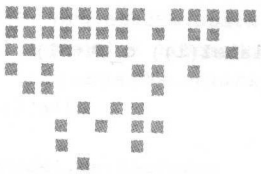
scala> c_dt
res8: Int = 927

// 统计决策树正确率
scala>  1.0*c_dt / t_count
res9: Double = 0.6250842886041807
```

可以看到，我们通过交叉验证得出最优参数，从而获得最佳模型，将这个过程使用流水线连接起来，方便了我们的工作。关于模型的优化，其实我们还有很多工作要做，后面第 11 章也给出了一定的优化思路和方法。

8.7　小结

本章就 Spark ML 中的分类模型进行了详细介绍，包括逻辑回归、决策树、朴素贝叶斯模型的原理，同时介绍了分类模型的一些使用场景。通过流水线、网格参数以及交叉验证的方式，将整个机器学习过程规范化、标准化、流程化。



Chapter 9 第 9 章

构建 Spark ML 回归模型

回归模型属于监督式学习，每个个体都有一个与之相关联的实数标签，并且我们希望在给出用于表示这些实体的数值特征后，所预测出的标签值可以尽可能接近实际值。

回归算法是试图采用对误差的衡量来探索变量之间关系的一类算法。回归算法是统计机器学习的利器。在机器学习领域，人们说起回归，有时候是指一类问题，有时候是指一类算法，这一点常常会使初学者感到困惑。常见的回归算法包括：普通最小二乘法（Ordinary Least Square，OLS），它使用损失函数是平方损失函数（$\frac{1}{2}\left(w^T x-y\right)^2$），简单的预测就是 $y=w^T x$，标准的最小二乘回归不使用正则化，这就意味着数据中异常数据点非常敏感，因此，在实际应用中经常使用一定程度的正则化（目的避免过拟合、提供泛化能力）。线性回归在应用 L2 正则化时通常称为岭回归（Ridge Regression），应用 L1 正则化时称为 Lasso 回归。L1、L2 正则化实际上就是在原来损失函数基础上添加一个约束，限制 w 的大小，实际为了降低损失函数对 X 的敏感程度。

$$\min_w \frac{1}{2}\left\|y-X_w\right\|^2, \text{s.t.}\left\|w\right\|_1 < \theta \text{（Lasso 回归）}$$

$$\min_w \frac{1}{2}\left\|y-X_w\right\|^2, \text{s.t.}\left\|w\right\|_2 < \theta \text{（Ridge 回归）}$$

正则化方法是其他算法（通常是回归算法）的延伸，其核心目的是为了避免过拟合（overfiting）。正则化方法通常对简单模型予以奖励而对复杂算法予以惩罚。常见的算法包括：Ridge Regression、Least Absolute Shrinkage and Selection Operator（LASSO）以及弹性网络（Elastic Net）。通过正则化方法有利于避免过拟合情况。下图为一份根据房屋面积预测销售价的示意图，从图 9-1 中可以看出，左侧图为欠拟合，右侧图存在过拟合，中间这

个图的泛化能力较好。

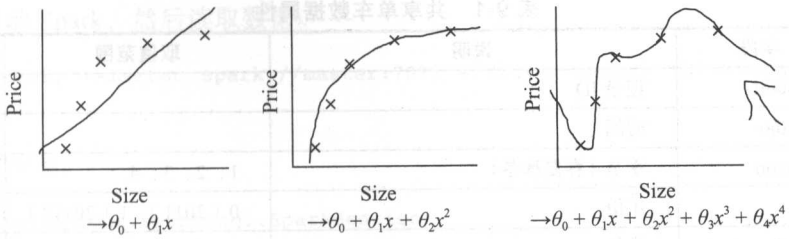


图 9-1　模型泛化能力

本章主要介绍 Spark ML 中的回归模型，以回归分析中常用决策树回归、线性回归为例，对共享单车租赁的情况进行预测，其中介绍了一些特征转换、特征选择、交叉验证等方法的具体使用，主要内容包括：

- ❑ 回归模型简介；
- ❑ 把数据加载到 HDFS，Spark 读取 HDFS 中的数据；
- ❑ 探索特征及其分布信息；
- ❑ 预处理数据；
- ❑ 把 Pipeline 的多个 Stage 组装到流水线上；
- ❑ 模型优化。

9.1　回归模型简介

ML 目前支持回归模型有：

- ❑ Linear regression（线性回归）
- ❑ Generalized linear regression（广义线性回归）
- ❑ Decision tree regression（决策树的回归）
- ❑ Random forest regression（随机森林回归）
- ❑ Gradient-boosted tree regression（梯度提高树回归）
- ❑ Survival regression（生存回归）
- ❑ Isotonic regression（保序回归）

9.2　数据加载

本章选择共享单车的数据集，这个数据集记录了共享自行车系统每小时自行车的出租次数。另外还包括日期、时间、天气、季节和节假日等相关信息（数据集下载地址：http://archive.ics.uci.edu/ml/datasets/Bike+Sharing+Dataset），其业务与我们平时租用摩拜单车类

似。表 9-1 为数据集的特征及属性。

表 9-1 共享单车数据属性

序号	字段	说明	取值范围	将忽略的项
1	instant	记录 ID		√
2	dteday	时间		√
3	season	季节（春夏秋冬）	1、2、3、4	
4	yr	年份	0（2011）、1（20112）	
5	mnth	月份	1～12	
6	hr	时间	0～23	
7	holiday	是否节假日	0、1	
8	weekday	周几	0～6	
9	workingday	是否工作日	0、1	
10	weathersit	天气类型	1、2、3、4	
11	temp	气温		
12	atemp	体感温度		
13	hum	湿度		
14	windspeed	风速		
15	casual	临时用户租车量		√
16	registered	注册用户租车量		√
17	cnt	目标变量，每小时的自行车租用量，等于 casual+registered		

查看数据大致情况：

```
#### 查看文件前 3 行数据
$ head -3 hour.csv
instant,dteday,season,yr,mnth,hr,holiday,weekday,workingday,weathersit,temp,atem
p,hum,windspeed,casual,registered,cnt
1,2011-01-01,1,0,1,0,0,6,0,1,0.24,0.2879,0.81,0,3,13,16
2,2011-01-01,1,0,1,1,0,6,0,1,0.22,0.2727,0.8,0,8,32,40
### 查看文件记录总数
$ wc -l hour.csv
17380 hour.csv
### 查看文件列数
cat hour.csv | head -1 | awk -F ',' '{print NF}'
17
```

从数据集前 3 行的数据可以看出，第一行为标题，其他为租赁数据，共有 17 个字段和 17380 条记录。

把数据文件 hour.csv 复制到 HDFS 上。

```
$ hadoop fs -put hour.csv  /home/hadoop/data
```

以独立模式启动 Spark，然后读取数据。

```
$ spark-shell --master spark://master:7077 --driver-memory 1G --total-executor-
cores 4
```

导入需要使用的类。

```
import  org.apache.spark.sql.SparkSession
import  org.apache.spark.sql.Row
import  org.apache.spark.sql.DataFrame
import  org.apache.spark.sql.Dataset
import  org.apache.spark.ml.Pipeline
import  org.apache.spark.ml.evaluation.RegressionEvaluator
import  org.apache.spark.ml.linalg.Vectors
import   org.apache.spark.ml.feature.{IndexToString, StringIndexer,
VectorIndexer,VectorAssembler}
import  org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}
import  org.apache.spark.ml.regression.DecisionTreeRegressionModel
import  org.apache.spark.ml.regression.DecisionTreeRegressor
import  org.apache.spark.ml.regression.LinearRegression
import   org.apache.spark.ml.regression.{RandomForestRegressionModel,
RandomForestRegressor}
import  org.apache.spark.ml.evaluation.MulticlassClassificationEvaluator
import  org.apache.spark.sql.types._
import  org.apache.spark.sql.functions._
```

读取数据，第一行为列名。

```
val rawdata = spark.read.format("csv").option("header", true).load("hdfs://
master:9000/home/hadoop/data/hour.csv")
```

查看前 4 行样本数据。

```
rawdata.show(4)
+-------+----------+------+---+----+---+-------+-------+----------+----------+----+------+----+---------+------+----------+---+
|instant|    dteday|season| yr|mnth| hr|holiday|weekday|workingday|weathersit|temp| atemp| hum|windspeed|casual|registered|cnt|
+-------+----------+------+---+----+---+-------+-------+----------+----------+----+------+----+---------+------+----------+---+
|      1|2011-01-01|     1|  0|   1|  0|      0|      6|         0|         1|0.24|0.2879|0.81|      0.0|     3|        13| 16|
|      2|2011-01-01|     1|  0|   1|  1|      0|      6|         0|         1|0.22|0.2727| 0.8|      0.0|     8|        32| 40|
|      3|2011-01-01|     1|  0|   1|  2|      0|      6|         0|         1|0.22|0.2727| 0.8|      0.0|     5|        27| 32|
|      4|2011-01-01|     1|  0|   1|  3|      0|      6|         0|         1|0.24|0.2879|0.75|      0.0|     3|        10| 13|
+-------+----------+------+---+----+---+-------+-------+----------+----------+----+------+----+---------+------+----------+---+
```

9.3　探索特征分布

Spark 读取数据后，我们就可以对数据进行探索和分析，首先查看前 4 行样本数据：

```
rawdata.show(4)
+-------+----------+------+---+----+---+-------+-------+----------+----------+----+------+----+---------+------+----------+---+
|instant|    dteday|season| yr|mnth| hr|holiday|weekday|workingday|weathersit|temp| atemp| hum|windspeed|casual|registered|cnt|
+-------+----------+------+---+----+---+-------+-------+----------+----------+----+------+----+---------+------+----------+---+
|      1|2011-01-01|     1|  0|   1|  0|      0|      6|         0|         1|0.24|0.2879|0.81|      0.0|     3|        13| 16|
|      2|2011-01-01|     1|  0|   1|  1|      0|      6|         0|         1|0.22|0.2727| 0.8|      0.0|     8|        32| 40|
|      3|2011-01-01|     1|  0|   1|  2|      0|      6|         0|         1|0.22|0.2727| 0.8|      0.0|     5|        27| 32|
|      4|2011-01-01|     1|  0|   1|  3|      0|      6|         0|         1|0.24|0.2879|0.75|      0.0|     3|        10| 13|
+-------+----------+------+---+----+---+-------+-------+----------+----------+----+------+----+---------+------+----------+---+
```

查看 rawdata 的数据结构。

```
rawdata.printSchema
root
 |-- instant: string (nullable = true)
 |-- dteday: string (nullable = true)
 |-- season: string (nullable = true)
 |-- yr: string (nullable = true)
 |-- mnth: string (nullable = true)
 |-- hr: string (nullable = true)
 |-- holiday: string (nullable = true)
 |-- weekday: string (nullable = true)
 |-- workingday: string (nullable = true)
 |-- weathersit: string (nullable = true)
 |-- temp: string (nullable = true)
 |-- atemp: string (nullable = true)
 |-- hum: string (nullable = true)
 |-- windspeed: string (nullable = true)
 |-- casual: string (nullable = true)
 |-- registered: string (nullable = true)
 |-- cnt: string (nullable = true)
```

目前这些数据的字段都是字符型，后续需要转换为数值型。

查看主要字段的统计信息。

```
rawdata.describe("dteday","holiday","weekday","temp").show()
+-------+----------+--------------------+-----------------+--------------------+
|summary|    dteday|             holiday|          weekday|                temp|
+-------+----------+--------------------+-----------------+--------------------+
|  count|     17379|               17379|            17379|               17379|
|   mean|      null|0.028770355026181024|3.003682605443351| 0.4969871684216586|
| stddev|      null|  0.1671652763843717|2.005771456110986|0.19255612124972202|
|    min|2011-01-01|                   0|                0|                0.02|
|    max|2012-12-31|                   1|                6|                   1|
+-------+----------+--------------------+-----------------+--------------------+
```

其中有很多字段是类型，如果使用回归算法时，需要通过 OneHotEncoder 把数据转换

为二元向量，对一些字段或特征进行规范化。

通过 pyspark 可以画出主要特征的重要程度。根据图 9-2 提供的信息，可知，特征 casual、registered 的贡献比较低，实际上 cnt=（casual+registered），故可考虑不使用这两个特征，另外记录 ID（instant）与日期（dteday）也对标识影响不大或无影响，可以过滤掉这 4 个特征（即：instant、dteday、casual、registered），最后剩下 1 个标特征，14 个预测特征。

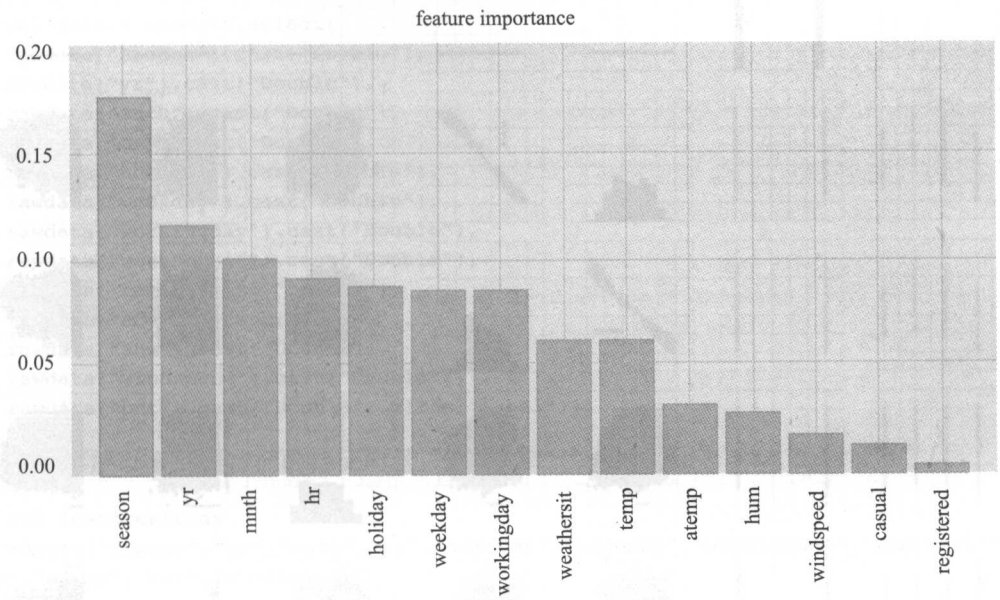


图 9-2　各特征的重要性

通过 pyspark 可以画出其中一些特征的分布情况：

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

df=pd.read_csv('/home/hadoop/data/bike/hour.csv',header=0)
sns.set(style='whitegrid',context='notebook')
cols=['season','yr','temp','atemp','hum','windspeed','cnt']
sns.pairplot(df[cols],size=2.5)
plt.show()
```

使用回归分析，人们往往会假设连续性特征服从正态分布，从图 9-3 的图形可知，'temp'、'atemp'、'hum'、'windspeed' 等 4 个特征基本满足正态分布，故使用回归分析时，已满足要求，但 cnt 不符合正态分布，这个特征是标签，我们先不对其标准化训练模型，然后在模型优化时，对该特征进行转换，使其接近正态分布，最后比较模型性能的变化。

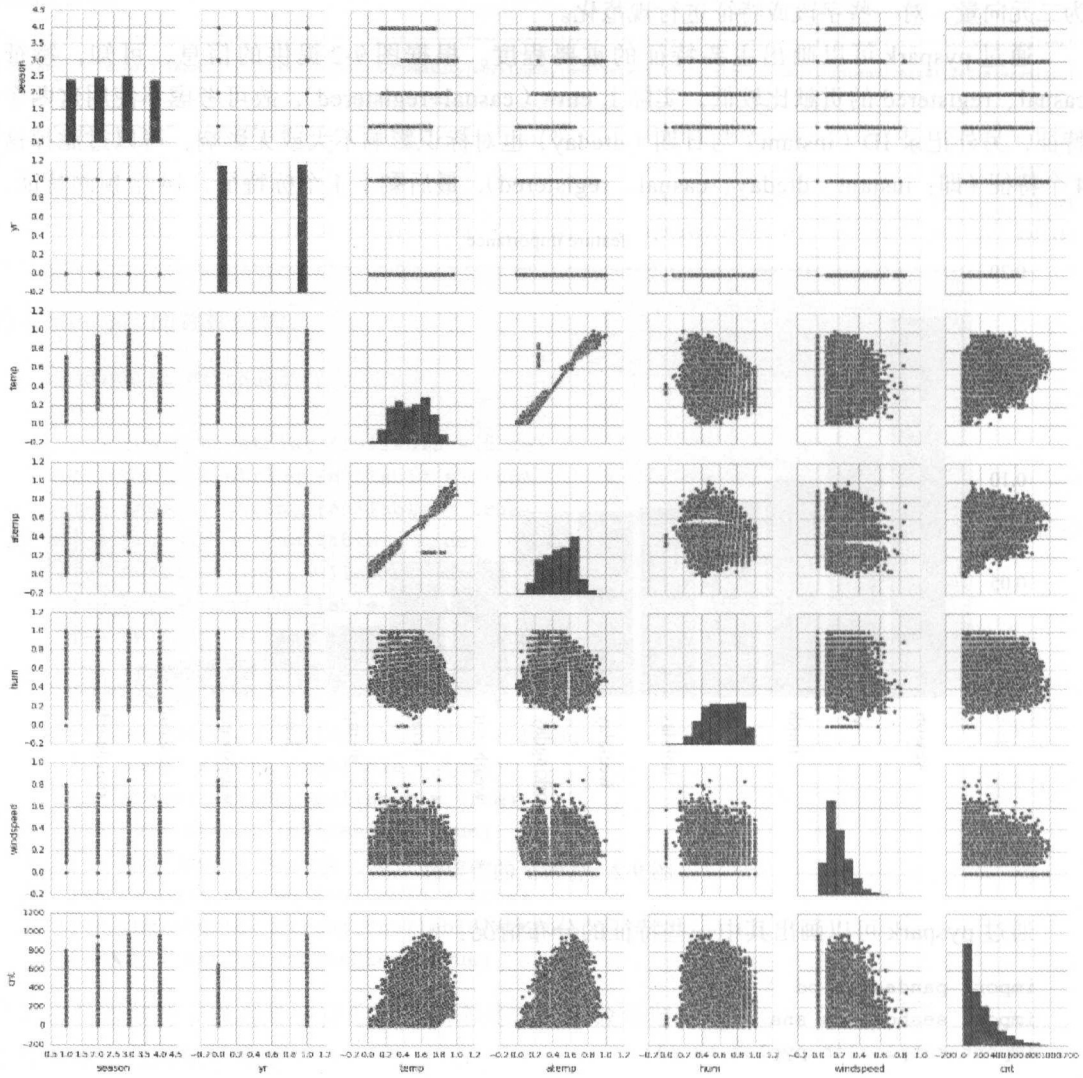


图 9-3 特征间的关系图

9.4 数据预处理

通过数据探索，我们发现数据的具体情况，为数据的预处理提供重要依据，当然，是否处理或如何处理，还需考虑具体使用的算法或模型，这里我们将分别采用决策树回归、线性回归两种方法，然后，比较两种方法的性能。在使用决策树回归时，除了做些特征选择外，其他特征转换将不考虑，但使用线性回归时需要考虑，否则将影响模型的性能。

先使用决策树回归来训练模型。

9.4.1　特征选择

首先把字符型的特征转换为数值类型，并过滤 instant、dteday、casual、registered 4 个无关或冗余特征。cnt 特征作为标志。

```
val data1= rawdata.select(
rawdata("season").cast("Double"),
rawdata("yr").cast("Double"),
rawdata("mnth").cast("Double"),
rawdata("hr").cast("Double"),
rawdata("holiday").cast("Double"),
rawdata("weekday").cast("Double"),
rawdata("workingday").cast("Double"),
rawdata("weathersit").cast("Double"),
rawdata("temp").cast("Double"),
rawdata("atemp").cast("Double"),
rawdata("hum").cast("Double"),
rawdata("windspeed").cast("Double"),
rawdata("cnt").cast("Double").alias("label"))
```

生成一个存放以上预测特征的特征向量：

```
val featuresArray
=Array("season","yr","mnth","hr","holiday","weekday","workingday","weathersit","
temp","atemp","hum","windspeed")
```

把原数据组合成特征向量 features：

```
val assembler = new
VectorAssembler().setInputCols(featuresArray).setOutputCol("features")
```

9.4.2　特征转换

这些特征大部分是分类特征，有些值是连续型，如气温、湿度等特征，使用决策树回归时，技术上可以通过给定类别个数的最大值（如 24），自动识别哪些特征作为类别特征，哪些作为连续性特征。这里把不同值小于或等于 24 的特征作为类别特征，大于 24 的视为连续性特征，并对分类特征索引化或数值化。

使用决策树回归算法前，我们对类别特征进行索引化或数值化。

```
val featureIndexer = new
VectorIndexer().setInputCol("features").setOutputCol("indexedFeatures").
setMaxCategories(24)
```

如果使用线性回归算法，我们需要对类别特征使用 OneHotEncoder（其具体使用可参考第 4 章），把它们转换为二元向量（或 one-Hot 编码）。

对前 8 个类别字段或特征转换为二元向量。

```
val data2= new OneHotEncoder().setInputCol("season").setOutputCol("seasonVec")
val data3= new OneHotEncoder().setInputCol("yr").setOutputCol("yrVec")
val data4= new OneHotEncoder().setInputCol("mnth").setOutputCol("mnthVec")
val data5= new OneHotEncoder().setInputCol("hr").setOutputCol("hrVec")
val data6= new OneHotEncoder().setInputCol("holiday").setOutputCol("holidayVec")
val data7= new OneHotEncoder().setInputCol("weekday").setOutputCol("weekdayVec")
val data8= new
OneHotEncoder().setInputCol("workingday").setOutputCol("workingdayVec")
val data9= new
OneHotEncoder().setInputCol("weathersit").setOutputCol("weathersitVec")
```

因 OneHotEncoder 不是 Estimator，这里我们对采用回归算法的数据另外进行处理，先建立一个流水线，把以上转换组装到这个流水线上。

```
val pipeline_en = new
Pipeline().setStages(Array(data2,data3,data4,data5,data6,data7,data8,data9))
val data_lr = pipeline_en.fit(data1).transform(data1)
```

把原来的 4 个及转换后的 8 个二元特征向量，拼接成一个 feature 向量。

```
val assembler_lr = new
VectorAssembler().setInputCols(Array("seasonVec","yrVec","mnthVec","hrVec", "ho
lidayVec","weekdayVec","workingdayVec","weathersitVec","temp","atemp","hum","windspe
ed")).setOutputCol("features_lr")
```

9.5 组装

1）将 data1 数据分为训练和测试集（30% 进行测试，种子设为 12）：

```
// 对 data1 数据集进行随机划分，这份数据用于决策模型
val Array(trainingData, testData) = data1.randomSplit(Array(0.7, 0.3),12)

// 对 data2 数据集进行随机划分，这份数据用于回归模型
val Array(trainingData_lr, testData_lr) = data_lr.randomSplit(Array(0.7, 0.3),12)
```

利用决策树回归模型训练可能要用到的参数：

- ❑ featuresCol：特征列名，默认值为 features；
- ❑ labelCol：标签列名，默认为 label；
- ❑ predictionCol：预测结果列名，默认为 prediction；
- ❑ maxDepth：树的最大深度，默认值为 5，
- ❑ maxBins：连续特征离散化的最大数量，以及选择每个节点分裂特征的方式，默认值为 32；
- ❑ minInstancesPerNode：分裂后自节点最少包含的实例数量，默认值为 1；

- minInfoGain：分裂节点时所需最小信息增益，默认值为 0.0；
- maxMemoryInMB：分配给直方图聚合的最大内存，默认为 256MB，
- cacheNodeIds=False，
- checkpointInterval：设置检查点间隔（>=1），或不设置检查点（-1）默认值为 10；
- impurity：计算信息增益的准则，默认值为 variance；
- seed：随机种子，默认为 None，
- varianceCol：预测的有偏样本偏差的列名，默认无。

2）设置决策树回归模型参数：

```
val dt = new DecisionTreeRegressor()
.setLabelCol("label")
.setFeaturesCol("indexedFeatures")
.setMaxBins(64)
.setMaxDepth(15)
```

3）设置线性回归模型的参数：

```
val lr =new LinearRegression()
.setFeaturesCol("features_lr")
.setLabelCol("label")
.setFitIntercept(true)
.setMaxIter(20)
.setRegParam(0.3)
.setElasticNetParam(0.8)
```

4）把决策树回归模型涉及的特征转换及模型训练组装在一个流水线上：

```
val pipeline = new Pipeline().setStages(Array(assembler,featureIndexer, dt))
```

5）把线性回归模型涉及的特征转换、模型训练组装载一个流水上线：

```
val pipeline_lr= new Pipeline().setStages(Array(assembler_lr,lr))
```

6）训练模型：

```
// 训练决策树回归模型
val model = pipeline.fit(trainingData)
// 训练线性回归模型
val lrModel = pipeline_lr.fit(trainingData_lr)
```

7）作出预测：

```
// 预测决策树回归的值
val predictions = model.transform(testData)
// 预测线性回归模型的值
val predictions_lr = lrModel.transform(testData_lr)
```

8）评估模型：

RegressionEvaluator.setMetricName 可以定义四种评估器：rmse（默认）、mse、r^2、mae。

```
val evaluator =new RegressionEvaluator()
.setLabelCol("label")
.setPredictionCol("prediction")
.setMetricName("rmse")
// 决策树模型评估指标
val rmse = evaluator.evaluate(predictions)
//rmse: Double = 61.62409114645229
val rmse_lr = evaluator.evaluate(predictions_lr)
// rmse_lr: Double = 102.05406408259029
```

从以上使用不同模型情况看来，决策树性能稍好与线性回归，但这仅是粗糙的比较，下面使用模型选择中介绍的一些方法，对线性模型进行优化。

9.6 模型优化

从图 9-3 可知，temp 特征与 atemp 特征线性相关，而且从图 9-2 可知，atemp 的贡献度较小，所以我们将过滤该特征。

```
    val assembler_lr1 = new
    VectorAssembler().setInputCols(Array("seasonVec","yrVec","mnthVec","hrVec", "ho
lidayVec","weekdayVec","workingdayVec","weathersitVec","temp","hum","windspeed")).
setOutputCol("features_lr1")
```

对 label 标签特征进行转换，使其更接近正态分布，这里我们 SQLTransformer 转换器，其具体使用可参考第 4 章。

```
// 导入需要的包
import org.apache.spark.ml.feature.SQLTransformer
// 对特征 label 进行 SQRT 运行
val sqlTrans = new SQLTransformer().setStatement(
  "SELECT *, SQRT(label) as label1 FROM __THIS__")
```

这里我们利用训练验证划分法对线性回归模型进行优化，对参数进行网格化，将数据集划分为训练集、验证集和测试集。

1）导入需要用到的包。

```
import org.apache.spark.ml.tuning.{ParamGridBuilder, TrainValidationSplit}
```

2）建立模型，预测 label1 的值，设置线性回归参数。

```
val lr1 = new LinearRegression()
.setFeaturesCol("features_lr1")
.setLabelCol("label1")
.setFitIntercept(true)
```

3）设置流水线，以便将特征组合、特征值优化、模型训练等任务组装到这条流水线上。

```
val pipeline_lr1 = new Pipeline().setStages(Array(assembler_lr1,sqlTrans,lr1))
```

4）建立参数网格。

```
val paramGrid = new ParamGridBuilder()
.addGrid(lr1.elasticNetParam, Array(0.0, 0.8, 1.0))
.addGrid(lr1.regParam,Array(0.1,0.3,0.5))
.addGrid(lr1.maxIter, Array(20, 30))
.build()
```

5）选择（prediction, label1），计算测试误差。

```
val evaluator_lr1 =new RegressionEvaluator()
.setLabelCol("label1")
.setPredictionCol("prediction")
.setMetricName("rmse")
// 利用交叉验证方法
val trainValidationSplit = new TrainValidationSplit()
  .setEstimator(pipeline_lr1)
  .setEvaluator(evaluator_lr1)
  .setEstimatorParamMaps(paramGrid)
  .setTrainRatio(0.8)
```

6）训练模型并自动选择最优参数。

```
val lrModel1 = trainValidationSplit.fit(trainingData_lr)
```

7）查看模型全部参数。

```
lrModel1.getEstimatorParamMaps.foreach { println }    // 参数组合
lrModel1.getEvaluator.extractParamMap()  // 查看评估参数
lrModel1.getEvaluator.isLargerBetter
```

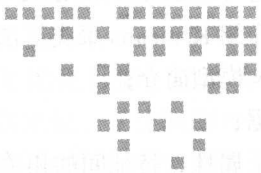
8）用最好的参数组合，做出预测。

```
val predictions_lr1 = lrModel1.transform(testData_lr)
val rmse_lr1 = evaluator_lr1.evaluate(predictions_lr1)
//rmse_lr1: Double = 3.1354674045018514
// 显示转换后特征值的前 5 行信息
predictions_lr1.select("features_lr1","label","label1","prediction").show(5)
// 结果显示如下：
+--------------------+-----+------------------+------------------+
|       features_lr1|label|            label1|        prediction|
+--------------------+-----+------------------+------------------+
|(55,[1,4,6,17,40,...| 39.0| 6.244997998398398| 2.544732781830004|
|(55,[1,4,6,17,40,...|  7.0|2.6457513110645907|1.1823933720401953|
|(55,[1,4,6,17,40,...|  5.0|  2.23606797749979|1.3641560005748419|
|(55,[1,4,6,17,40,...|  7.0|2.6457513110645907|1.7674507231166494|
|(55,[1,4,6,17,40,...| 12.0|3.4641016151377544|1.5020350291356124|
```

看了对标签特征进行转换、利用网格参数及训练验证划分等优化方法，从 102 下降到 3 左右，效果比较明显。

9.7 小结

本章主要介绍 Spark ML 的线性回归模型、决策树回归模型，对共享单车的租赁信息进行预测，由于很多数据不规范，因此，对原数据进行了二元向量转换、对类别数据索引化，然后把这些转换组装到流水线上，在训练集上训练模型，在测试集上进行预测，最后，评估指标对模型进行优化。



第 10 章 Chapter 10

构建 Spark ML 聚类模型

前面我们介绍了推荐、分类、回归等模型，这些模型属于监督学习，在训练模型时，都提供目标值或标签数据，根据目标值训练模型，然后根据模型对测试数据或新数据进行推荐、分类或预测。

但实际数据有很多是没有标签数据或者预先标签很难的，但我们又希望或需要从这些数据中提炼一些规则或特征等，如识别异常数据、对客户进行分类等，这类问题就属于无监督学习。

聚类是一种无监督学习，它与分类不同，聚类所要求划分的类是未知的。

聚类算法的思想就是物以类聚的思想，相同性质的点在空间中表现得较为紧密和接近，主要用于数据探索与异常检测。

聚类分析是一种探索性的分析，在分类的过程中，人们不必事先给出一个分类的标准，它能够从样本数据出发，自动进行分类。聚类分析也有很多方法，使用不同方法往往会得到不同的结论。从实际应用的角度看，聚类分析是数据挖掘的主要任务之一。而且聚类能够作为一个独立的工具获得数据的分布状况，观察每一族数据的特征，集中对特定的族集合作进一步地分析。聚类分析还可以作为其他算法（如分类和推荐等算法）的预处理步骤。

聚类是机器学习中一种重要方法，一般机器学习中都有，当然 Spark 也不例外，Spark 目前支持的聚类算法有：

- K 均值（K-means）
- 三层贝叶斯概率模型（Latent Dirichlet allocation，LDA）
- 二分 K 均值（Bisecting K-means）

- 高斯混合（Gaussian Mixture Model，GMM）

本章主要介绍 K-means 聚类算法，主要内容包括：

- K-means 模型简介；
- 加载数据；
- 探索特征属性、特征间的相关性等；
- 预处理数据；
- 将 Pipeline 的多个 Stages 组装在流水线上；
- 模型调优。

10.1 K-means 模型简介

作为经典的聚类算法，一般的机器学习框架里都有 K-means，Spark 自然也不例外。不过 Spark 中的 K-means，除有一般 K-means 的特点外，还进行了如下的优化：

1. 选择合适的 K 值

k 的选择是 K-means 算法的关键。Spark 在 KMeansModel 里实现了 computeCost 方法，这个方法通过计算数据集中所有的点到最近中心点的平方和来衡量聚类的效果。一般来说，同样的迭代次数，这个 cost 值越小，说明聚类的效果越好。但在实际使用过程中，必须还要考虑聚类结果的可解释性，不能一味地选择 cost 值最小的那个 k。比如我们考虑极限情况，数据集有 n 个点，如果令 $k=n$，每个点都是聚类中心，每个类都只有一个点，此时 cost 值最小为 0。但是这样的聚类结果显然是没有实际意义的。

2. 选择合适的初始中心点

大部分迭代算法都对初始值很敏感，K-means 也是如此。Spark 在初始中心点的选择上，使用了 K-means++ 的算法。想要详细了解 K-means++ 的同学们，可以参考 K-means++ 在 wiki 上的介绍：https://en.wikipedia.org/wiki/K-means%2B%2B。

K-means++ 的基本思想是使初始中心店的相互距离尽可能远。为了实现这个初衷，采取如下步骤：

1）从初始数据集中随机选择一个点作为第一个聚类中心点。

2）计算数据集中所有点到最近一个中心点的距离 D(x) 并存在一个数组里，然后将所有这些距离加起来得到 Sum(D(x))。

3）然后再取一个随机值，用权重的方式计算下一个中心点。具体的实现方法：先取一个在 Sum(D(x)) 范围内的随机值，然后令 Random - = D(x)，直至 Random <= 0，此时这个 D(x) 对应的点为下一个中心点。

4）重复 2、3 步直到 k 个聚类中心点被找出。

5）利用找出的 k 个聚类中心点，执行标准的 K-means 算法。

类似大多数机器学习模型，K-means 聚类需要数值向量作为输入，于是用于分类和回归的特征提取和变换方法也适用于聚类。

K-means 和最小方差回归一样使用方差函数作为优化目标，因此容易受到离群值（outlier）和较大方差的特征影响。对于回归和分类问题来说，上述问题可以通过特征的归一化和标准化来解决，同时可能有助于提升性能。但是某些情况我们可能不希望数据被标准化，比如根据某个特定的特征找到对应的类族。

10.2　数据加载

这里我们以某批发经销商的客户对不同产品的年度消费支出为例（数据来源 http://archive.ics.uci.edu/ml/datasets/Wholesale+customers），具体信息如表 10-1 所示。

表 10-1　数据源结构说明

字段	含义（一年内的消费）	备注
Channel	表示购买的渠道	1- 团购、2- 零售购买
Region	表示顾客所属的地区	1- 里斯本、2- 波尔图、3- 其他
Fresh	在新鲜产品的年度支出上	
Milk	在奶制产品上的消费	
Grocery	在零食上的消费	
Frozen	在冷冻食品上的消费	
Detergents_Paper	在洗涤用品和纸上的消费	
Delicassen	在熟食上的消费	

读取 HDFS 中的数据。

```
// 导入需要的类
import org.apache.spark.ml.clustering.KMeans
import org.apache.spark.ml.feature.VectorAssembler
import org.apache.spark.ml.feature.OneHotEncoder
import org.apache.spark.ml.feature.StandardScaler
import org.apache.spark.ml.{Pipeline, PipelineModel}

// 通过 spark.read, 读取 HDFS 中的数据
val rawdata = spark.read.format("csv").option("header",
true).load("hdfs://master:9000/home/hadoop/data/customers_sale.csv")
```

10.3　探索特征的相关性

数据加载后，首先我们看一下数据集的统计信息，看是否有缺失值、各特征的统计信息等。

```
//数据的样本信息
rawdata.show(3)
+-------+------+-----+----+-------+------+----------------+----------+
|Channel|Region|Fresh|Milk|Grocery|Frozen|Detergents_Paper|Delicassen|
+-------+------+-----+----+-------+------+----------------+----------+
|      2|     3|12669|9656|   7561|   214|            2674|      1338|
|      2|     3| 7057|9810|   9568|  1762|            3293|      1776|
|      2|     3| 6353|8808|   7684|  2405|            3516|      7844|
+-------+------+-----+----+-------+------+----------------+----------+
//查看数据结构
rawdata.printSchema()
root
  |-- Channel: string (nullable = true)
  |-- Region: string (nullable = true)
  |-- Fresh: string (nullable = true)
  |-- Milk: string (nullable = true)
  |-- Grocery: string (nullable = true)
  |-- Frozen: string (nullable = true)
  |-- Detergents_Paper: string (nullable = true)
  |-- Delicassen: string (nullable = true)
```

从以上分析可以看出，rawdata 数据集总记录数为 440 条，最大与最小值相差不大，已统计的特征来看，没有缺失值，数据类型为字符型，这点需要在预处理中转换为 Double 型。

利用 pyspark 我们可以画出这些特征间的相关性，这里使用 pearson's r，相关系统在 [-1,1] 之间，如果 r=1，表示特征完全正相关；r=0，表示不存在关系；r=−1，表示特征完全负相关。

实现代码：

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

df=pd.read_csv('/home/hadoop/data/customer_sale/customers_sale.csv',header=0)
cols=['Channel','Region','Fresh','Milk','Grocery','Frozen','Detergents_
Paper','Delicassen']
cm=np.corrcoef(df[cols].values.T)

sns.set(font_scale=1.2)
hm=sns.heatmap(cm,cbar=True,annot=True,square=True,fmt='.2f',annot_kws={'size':1
5},yticklabels=cols,xticklabels=cols)
plt.show()
plt.savefig('sale_corr.png')
```

代码实现结果如图 10-1 所示。

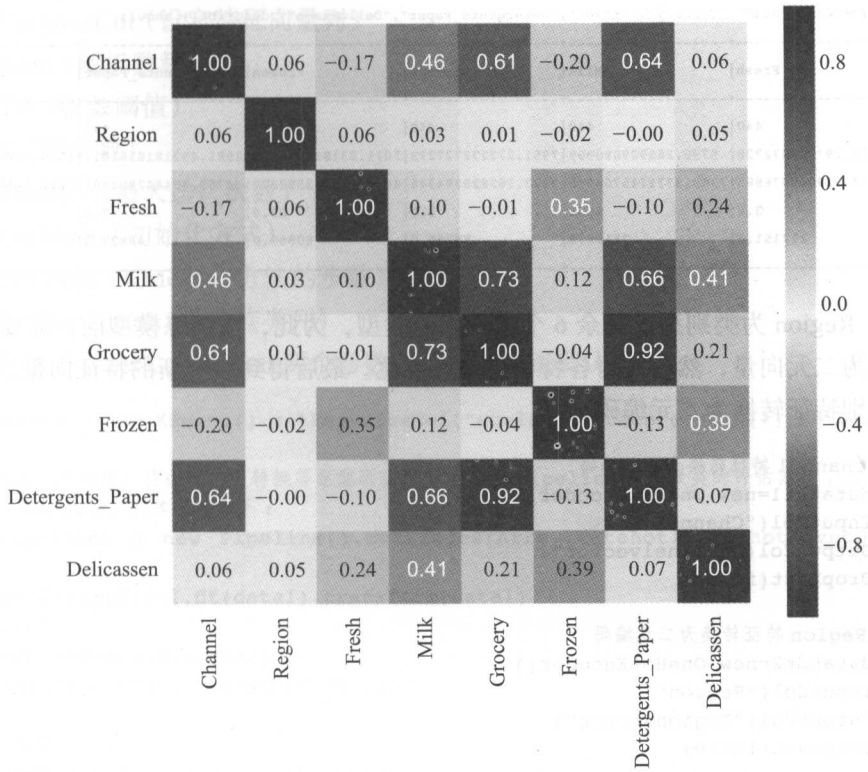


图 10-1　特征间的相关性

从图 10-1 可以看出，零食上的消费（Grocery）与在洗涤用品和纸上的消费（Detergents_Paper）相关性最大（r=0.92）；零食上的消费（Grocery）与在奶制产品上的消费（Milk）相关性也较大（r=0.73）；另外，渠道（Channel）与零食上的消费、洗涤用品和纸上的消费有一定的依赖关系。

10.4　数据预处理

通过数据探索，发现数据需要又字符转换为数值型，并缓存。

```
val data1= rawdata.select(
rawdata("Channel").cast("Double"),
rawdata("Region").cast("Double"),
rawdata("Fresh").cast("Double"),
rawdata("Milk").cast("Double"),
rawdata("Grocery").cast("Double"),
rawdata("Frozen").cast("Double"),
rawdata("Detergents_Paper").cast("Double"),
rawdata("Delicassen").cast("Double")).cache()
```

查看数据的统计信息：

```
datal.select("Fresh","Milk","Grocery","Frozen","Detergents_Paper","Delicassen").describe().show()
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
|summary|            Fresh|             Milk|          Grocery|           Frozen| Detergents_Paper|        Delicassen|
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
|  count|              440|              440|              440|              440|              440|              440|
|   mean|12000.297727272728| 5796.265909090909|7951.277272727273|3071.931818181818|2881.4931818181817|1524.8704545454545|
| stddev|12647.328865076885|7380.3771745708445|9503.162828994346|4854.673332592367| 4767.854447904201|2820.1059373693965|
|    min|              3.0|             55.0|              3.0|             25.0|              3.0|              3.0|
|    max|         112151.0|          73498.0|          92780.0|          60869.0|          40827.0|          47943.0|
+-------+-----------------+-----------------+-----------------+-----------------+-----------------+-----------------+
```

Channel、Region 为类别型，其余 6 个字段为连续型，为此，在训练模型前，需要对类别特征先转换为二元向量，然后，对各特征进行规范化。最后得到一个新的特征向量。

将类别特征转换为二元编码：

```
// 把 channel 特征转换为二元编码
val datahot1=new OneHotEncoder()
.setInputCol("Channel")
.setOutputCol("Channelvector")
.setDropLast(false)

// 把 Region 特征转换为二元编码
val datahot2=new OneHotEncoder()
.setInputCol("Region")
.setOutputCol("Regionvector")
.setDropLast(false)
```

把新生成的 2 个特征及原来的 6 个特征组成一个特征向量：

```
val featuresArray
=Array("Channelvector","Regionvector","Fresh","Milk","Grocery","Frozen","Deterge
nts_Paper","Delicassen")
```

把原数据组合成特征向量 features：

```
val vecDF = new VectorAssembler()
.setInputCols(featuresArray)
.setOutputCol("features")
```

对特征进行规范化：

```
val scaledDF = new StandardScaler()
  .setInputCol("features")
  .setOutputCol("scaledFeatures")
  .setWithStd(true)
  .setWithMean(false)
```

10.5 组装

ML 包下的 KMeans 方法有如下参数可供设置：

- setFeaturesCol（设定特征向量列）
- setSeed（随机数种子）
- setTol（收敛阈值）
- setK（族个数）
- setMaxIter（最大迭代次数）
- setInitMode（初始化方式）
- setInitSteps（KMeans|| 方法的步数）
- setPredictionCol（设置族类列）

这里我们只使用了 setK、setSeed 两个参数，其余的使用默认值。

```
    val kmeans = new KMeans().setFeaturesCol("scaledFeatures").setK(4).setSeed(123)

    //把转换二元向量、特征规范化转换等组装到流水线上，因 pipeline 中无聚类的评估函数，因此，这里流水线中不纳入 K-means。具体实现如下：
    val pipeline1 = new Pipeline().setStages(Array(datahot1,datahot2,vecDF,scaledDF))
    val data2=pipeline1.fit(data1).transform(data1)
    //训练模型
    val model=kmeans.fit(data2)
    val results = model.transform(data2)

    //评估模型
    val WSSSE = model.computeCost(data2)
    println(s"Within Set Sum of Squared Errors = $WSSSE")
    //显示聚类结果
    println("Cluster Centers: ")
    model.clusterCenters.foreach(println)
    results.collect().foreach(row => {println( row(10) + " is predicted as cluster " + row(11))})
    //部分结果
    [0.0,0.0,1.0,0.0,0.0,0.0,1.0,11867.0,3327.0,4814.0,1178.0,3837.0,120.0] is
    predicted as cluster
    [0.0,0.0,2.1365167114232353,0.0,0.0,0.0,2.220261941072331,0.9383008955170281,0.4507899693071525,0.5065681906777801,0.24265278408979063,0.8047644998237-361,0.0425515929773675]
    [0.0,0.0,1.0,0.0,0.0,0.0,1.0,16117.0,46197.0,92780.0,1026.0,40827.0,2944.0] is predicted as cluster
    [0.0,0.0,2.1365167114232353,0.0,0.0,0.0,2.220261941072331,1.2743402319919055,6.259436192390298,9.763065378289248,0.21134274743304346,8.562971132213624,1.0439324143780826]
    //当 k=4 的记录数
    results.select("scaledFeatures","prediction").groupBy("prediction").count.show()
    +----------+-----+
    |prediction|count|
    +----------+-----+
    |         1|   10|
    |         3|  136|
```

```
|          2|   64|
|          0|  230|
+-----------+-----+
// 由此可知，第 0,3 簇较大，1,2 比较小
results.select("scaledFeatures","prediction").filter(i=>i(1)==0).show(20)
val
result0=results.select("scaledFeatures","prediction").filter(i=>i(1)==0).
select("scaledFeatures")
```

10.6 模型优化

聚类模型中最重要的是参数 k 的选择，下面我们通过循环来获取哪个 k 值的性能最好。

```
val KSSE = (2 to 20 by 1).toList.map { k =>
val kmeans = new KMeans().setFeaturesCol("scaledFeatures").setK(k).setSeed(123)
val model = kmeans.fit(data2)
// 评估性能
val WSSSE = model.computeCost(data2)

// k,实际迭代次数,SSE,聚类类别编号，每类的记录数，类中心点
(k, model.getMaxIter, WSSSE, model.summary.cluster, model.summary.clusterSizes,
model.clusterCenters)
    }

// 显示 k、WSSSE 评估指标，并按指标排序
KSSE.map(x=>(x._1,x._3)).sortBy(x=>x._2).foreach(println)
// 显示结果
(20,635.6231456631109)
(19,674.1240263779249)
(18,696.2925462727684)
(17,747.697734807987)
(15,848.393503421027)
(16,878.8045714559038)
(14,932.4137349866897)
(13,988.2458378719449)
(12,1026.9426528633646)
(11,1165.7468060138433)
(10,1201.1295734061587)
(9,1242.388169008257)
(8,1399.0770764839624)
(7,1523.4613624094593)
(6,1965.6551642041663)
(5,2405.5349119889274)
(4,2595.7328287620885)
(3,3123.9948271417393)
(2,3480.224930619828)
// 把该结果保存到 HDFS 上
```

```
KSSE.map(x=>(x._1,x._3)).sortBy(x=>x._2).toDF.write.save("/home/hadoop/data/
ksse")
```

以上数据可视化的图形如图 10-2 所示。

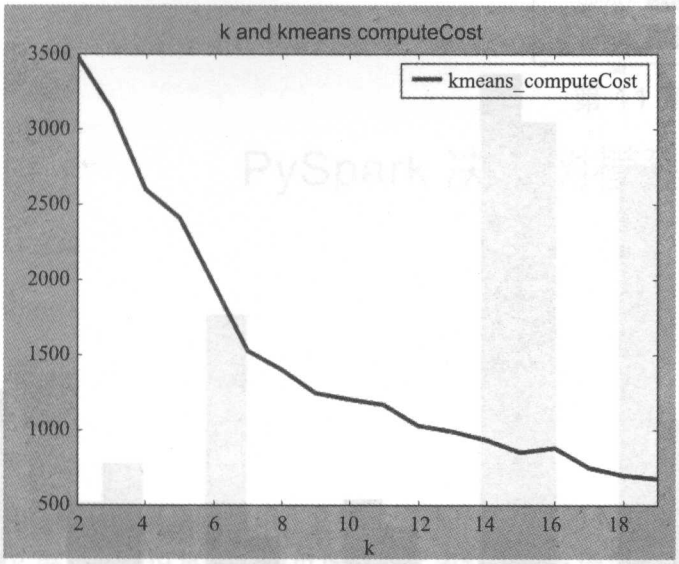


图 10-2　聚类模型中族 k 与评估指标的关系

从图 10-2 中不难看出，k<12 时，性能（computeCost）提升比较明显，k>12 后，逐渐变缓。所以 k 越大不一定越好，恰当才是重要的。

当 k=10 时，聚类结果如下：

```
+----------+-----+
|prediction|count|
+----------+-----+
|         1|   65|
|         6|   18|
|         3|   86|
|         5|   27|
|         9|    3|
|         4|    2|
|         8|    2|
|         7|   46|
|         2|   27|
|         0|  164|
+----------+-----+
```

图 10-3 为 k 取 10 时（族 0 对应的 channel 和 Region 分别为 1 和 3；族 3 对应的 channel 和 Region 分别为 2 和 3），前两大族的销售均值比较图，从图中可以看出，团购冷藏食品均值大于或接近零售冷藏食品均值。说明团购对冷藏食品量比较大。

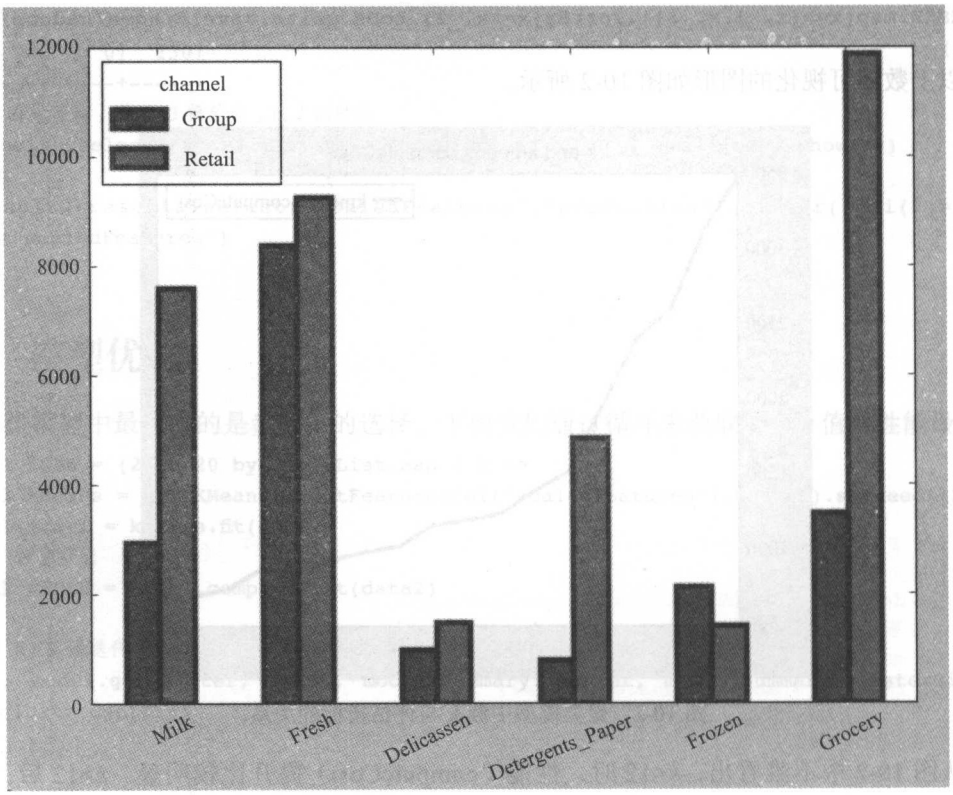


图 10-3 聚类模型中 k=10 时 0 和 3 族平均销售额对比

10.7 小结

本章主要介绍了用 Spark ML 中的聚类算法，对某地多种销售数据进行聚类分析，在分析前对数据集主要特征进行了相关性分析，并对类别数据进行二元向量化，对连续性数据进行规范和标准化，然后把这些 stages 组装在流水线上，在模型训练中，我们尝试不同 k 的取值，以便获取最佳族群数。

第 11 章 Chapter 11

PySpark 决策树模型

Spark 不但好用、而且还易用、通用，它提供多种开发语言的 API，除了 Scala 外，还有 Java、Python、R 等，可以说集成目前市场最有代表性的开发语言，使得 Spark 受众上升几个数据量级，同时也无形中降低了学习和使用的门槛，使得很多熟悉 Java、Python、R 的编程人员、数据分析师，也可以方便地利用 Spark 大数据计算框架来实现他们的大数据处理、机器学习等任务。

Python 作为机器学习中的利器，一直被很多开发者和学习者所推崇。除了开源、易学，以及简洁的代码风格的特性之外，Python 当中还有很多优秀的第三方库，为我们对数据进行处理、探索和模型的构建提供很大的便利，如 Pandas、Numpy、Scipy、Matplotlib、StatsModels、Scikit-Learn、Keras 等。Python 的强大还体现在它的与时俱进，它与大数据计算平台 Spark 的结合，可谓是强强联合、优势互补、相得益彰，这就有了现如今 Spark 当中一个重要分支——PySpark。其内部架构可参考图 11-1（该图取自 https://cwiki.apache.org/confluence/display/SPARK/PySpark+Internals?spm=5176.100239.0.0.eI85ij）。

PySpark 的 Python 解释器在启动时会同时启动一个 JVM，Python 解释器与 JVM 进程通过 socket 进行通信，在 python driver 端，SparkContext 利用 Py4J 启动一个 JVM 并产生一个 JavaSparkContext。Py4J 只使用在 driver 端，用于本地 python 与 Java SparkContext objects 的通信。大量数据的传输使用的是另一个机制。RDD 在 python 下的转换会被映射成 java 环境下 PythonRDD。在远端 worker 机器上，PythonRDD 对象启动一些子进程并通过 pipes 与这些子进程通信，以此 send 用户代码和数据。

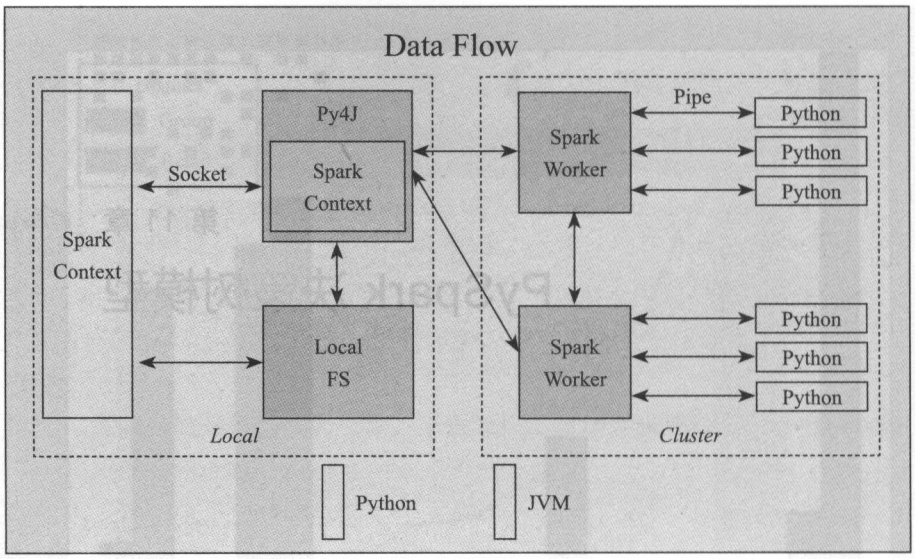


图 11-1 PySpark 架构图

本章就机器学习中的决策树模型，使用 PySpark 中的 ML 库，以及 IPython 交互式环境进行讲解。具体内容如下：

- 决策树简介；
- 数据加载；
- 数据探索；
- 创建决策树模型；
- 训练模型并进行预测；
- 利用交叉验证、网格参数等进行模型调优；
- 最后生成一个可执行 Python 脚本。

11.1 PySpark 简介

在 Spark 的官网上这么介绍 PySpark：“PySpark is the Python API for Spark”，也就是说 PySpark 其实是 Spark 为 Python 提供的编程接口。此外，Spark 还提供了关于 Scala、Java 和 R 的编程接口，关于 Spark 为 R 提供的编程接口（SparkR）将在第 12 章进行介绍。

与传统的 Spark 相比，PySpark 的优势体现在：

- 可以使用 Python 编程，并且可以使用 PySpark 直接运行 Python 的脚本；
- PySpark 具备了 Python 强大的可视化功能，这大大弥补了 Spark 无法定制高质量的可视化图形的缺陷；
- PySpark 将 IPython 整合到 Spark 中，强大的代码补全功能，提供了更好的交互式编

程体验。

与传统的 Python 相比，PySpark 的优势体现在：

- PySpark 的底层架构依旧是基于 HDFS 文件系统，这就意味着 PySpark 可以直接读取 HDFS 中的数据文件，而传统的 Python 读取 HDFS 中的文件需要借助 hdfs3 这个第三方库。同时 PySpark 可以直接读取 HBase 和 Hive 中的数据表；
- PySpark 中的 DataFrame 较 Pandas 中的 DataFrame 可以容纳更多的数据集，实现了真正意义上的大数据处理，并且提供了 schema，这是 Pandas 所不具有的；
- PySpark 支持 Hadoop，实现了分布式处理数据量大的文件，效率更加高效；
- PySpark 将机器学习的库已经进行封装，并且提供了流水线（Pipeline），大大方便了开发者的使用；
- PySpark 采用分布式并行计算框架，内建并行机制 parallelism，所有的数据和操作自动并行分布在各个集群结点上；
- PySpark 采用内存机制处理，可以将 RDD 或者 DataFrame 保存在内存中，减少 I/O 读取，使得数据处理更加高效；一般的 Python 只支持单机缓存。

PySpark 提供了 2 种机器学习的包，分别是 MLlib 和 ML。MLlib 是基于 RDD 的，从 Spark 2.0 开始，基于 RDD 的 API 进入维护模式（即不再增加任何新的特性），并预期于 3.0 版本的时候从 Spark 中移除。ML 提供了基于 DataFrames 高层次的 API，可以用来构建机器学习工作流（Pipeline）。ML Pipeline 弥补了原始的 MLlib 库的不足，向用户提供了一个基于 DataFrame 的机器学习工作流 API 套件。

11.2　决策树简介

决策树在机器学习中是很常见且经常使用的模型，它是一种强大的非概率模型，可以用来表达复杂的非线性模式和特征相互关系。其数据结构如表 11-1 所示。决策树在很多任务上表现出很好的性能，相对容易理解和解释，可以处理类属或者数值特征，同时不要求输入数据的归一化或者标准化。

表 11-1　数据结构

PID	age	income	student	credit_rating	Class:buys_computer
1	youth	high	no	fair	no
2	youth	high	no	excellent	no
3	middle_aged	high	no	fair	yes
4	senior	medium	no	fair	yes
5	senior	low	yes	fair	yes
6	senior	low	yes	excellent	no
7	middle_aged	low	yes	excellent	yes
8	youth	medium	no	fair	no

（续）

PID	age	income	student	credit_rating	Class:buys_computer
9	youth	low	yes	fair	yes
10	senior	medium	yes	fair	yes
11	youth	medium	yes	excellent	yes
12	middle_aged	medium	no	excellent	yes
13	middle_aged	high	yes	fair	yes
14	senior	medium	no	excellent	no

如图 11-2 所示，是使用决策树模型来评价用户是否购买电脑的模型结构。模型通过对不同特征值的判断，来预测用户是否会购买电脑。

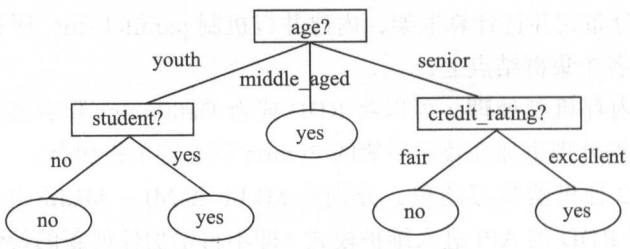


图 11-2 决策树结构

整个过程：首先从用户的年龄开始判断，如果用户是中年人，那么就可能会购买电脑；如果是年轻人，则进行下一步判断，如果是学生就可能会购买，否则不会购买。如果是年长者，那么根据信用评分来进行下一步的判断，如果是一般的，那么不会购买，如果是极好的，那么就可能购买。整个流程从根节点开始不断向下做判断，呈现树状的结构，关于根节点以及叶子节点的选择是通过信息熵进行计算而做出选择。

决策树算法是一种自上而下始于根节点（或特征）的方法，在每一个步骤中通过评估特征分裂的信息增益，最后选出分割数据集最优的特征。信息增益通过计算节点不纯度（即节点标签不相似或不同质的程度）减去分割后的两个子节点不纯度的加权和。对于分类任务，这里有两个评估方法用于选择最好的分割：基尼系数和熵。

关于决策树的原理，这里不再赘述。本章着重讨论的是，决策树的分类模型在 PySpark 中的应用。

11.3 数据加载

11.3.1 原数据集初探

这里的数据选择为某比赛的数据集，用来预测推荐的一些页面是短暂（昙花一现）还是长久（长时流行）。原数据集为 train.tsv，存放路径为 /home/hadoop/data/train.tsv。

先使用 shell 命令对数据进行试探性的查看，并做一些简单的数据处理。

1）查看前 2 行数据：

```
$ head -2 train.tsv

"url" "urlid" "boilerplate"    "alchemy_category"    "alchemy_category_score"
  "avglinksize"           "commonlinkratio_1" v      "commonlinkratio_2"
  "commonlinkratio_3"     "commonlinkratio_4"        "compression_ratio"
  "embed_ratio"           "framebased"     "frameTagRatio"    "hasDomainLink"
  "html_ratio"            "image_ratio"    "is_news"
  "lengthyLinkDomain"     "linkwordscore"  "news_front_page"
"non_markup_alphanum_characters" "numberOfLinks"    "numwords_in_url"
  "parametrizedLinkRatio" "spelling_errors_ratio"   "label"
"http://www.bloomberg.com/news/2010-12-23/ibm-predicts-holographic-calls-air-
breathing-batteries-by-2015.html" "4042" "{""title"":""IBM Sees Holographic Calls
Air Breathing Batteries ibm sees holographic calls, air-breathing
batteries"",""body"":""A sign stands outside the International Business Machines
Corp IBM Almaden Research Center campus in San Jose Cali    "8"
    ..............  "0.152941176" "0.079129575"    "0"
```

数据集中的第 1 行为标题（字段名）行，表 11-2 是一些字段的说明。紧接着的 22 列为各种数值和页面的所属类别。

最后一列为标签，用来标识网页是长久的还是暂时的。取值为 0 和 1，1 标识此页面是长久的，0 表示此页面是暂时的。

表 11-2　数据集主要字段说明

字段	说明
url	页面的 url
urlid	页面 ID
boilerplate	页面的文本内容
alchemy_category	页面所属类别
label	页面类别

2）查看文件记录总数：

```
$ cat train.tsv |wc -l
7396
```

结果显示共有：数据集一共有 7396 条数据。

3）由于 textFile 目前不好过滤标题行数据，为便于 spark 操作数据，需要先删除标题：

```
$ sed  1d train.tsv >train_noheader.tsv
```

4）将数据文件上传到 hdfs：

```
$ hdfs dfs -put train_noheader.tsv /data
```

5）查看是否成功：

```
hadoop@master:~/data$ hdfs dfs -ls /data
17/05/24 00:46:20 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Found 1 items
-rw-r--r--   1 hadoop supergroup   21972457 2017-05-24 00:46
/data/train_noheader.tsv
```

11.3.2 PySpark 的启动

以 spark Standalone 模式启动 Spark 集群，保证内存分配充足。

```
$ pyspark --master spark://master:7077 --driver-memory 4G --total-executor-cores 4
```

PySpark 的常见的启动指令如表 11-3 所示。

表 11-3 pyspark 启动参数说明

指令	说明
master	启动入口，常见的有：spark、mesos、yarn 以及 local
driver-memory	驱动内存，默认是 1024M
executor-memory	执行器内存，默认是 1G
conf	任意的 Spark 配置文件
total-executor-cores	执行器的总核心数

注：使用 pyspark --help 可以查看指令的详细帮助信息。



1）在 Spark 2.0 之前，启动 PySpark 的 IPython 交互式使用：

IPYTHON=1 IPYTHON_OPTS=" --pylab " pyspark --master spark://master:7077 --driver-memory 4G --total-executor-cores 4

2）在 Spark 2.0 以后版本，移除了 IPYTHON 和 IPYTHON_OPTS 指令。可以在配置文件中设置启动 PySpark 时默认 shell 环境为 IPython。

在 spark/bin/pyspark 文件中，以下地方做修改，设置 PySpark 默认启动为 IPython。

```
# Default to standard python interpreter unless told otherwise
if [[ -z "$PYSPARK_DRIVER_PYTHON" ]]; then
   PYSPARK_DRIVER_PYTHON="${PYSPARK_PYTHON:-ipython}"
fi
```

11.3.3 基本函数

这里对本章中需要用到函数和方法做一个简单的说明，如表 11-4 所示。

表 11-4 本章使用的一些函数或方法简介

函数	说明
sc.textFile(path)	读取路径中的数据文件，产生一个 RDD
rdd.take(n)	查看前 *n* 行数据
rdd.count()	统计 RDD 中的元素个数
rdd.countByKey()	按键进行统计个数
rdd.first()	查看数据集的第一行数据
rdd.collect()	将 RDD 中所有元素以列表的形式返回
Vectors.dense(*elements)	从列表或数值中创建一个密集矢量
spark.createDataFrame()	创建一个 DataFrame 对象

11.4　数据探索

1）通过 sc 对象的 textFile 方法，载入本地数据文件，创建 RDD：

```
In [1]: raw_data = sc.textFile("hdfs://master:9000/data/train_noheader.tsv")
```

2）查看第 1 行数据：

```
In [2]: raw_data.take(2)
Out[2]:[u'"http://www.bloomberg.com/news/2010-12-23/ibm-predicts-holographic-calls-air-breathing-batteries-by-2015.html"\t"4042"\t{""title"":""IBM Sees
    Holographic Calls Air Breathing Batteries ibm sees holographic calls, air-breathing batteries"",""body"":""A sign stands outside the International Business Machines Corp IBM Almaden Research Center campus in San Jose
    Cali..."\t...\t"8"\t"0.152941176"\t"0.079129575"\t"0"',
    u'"http://www.popsci.com/technology/article/2012-07/electronic-futuristic-starting-gun-eliminates-advantages-races"\t"8471"\t{""title"":""The Fully
    Electronic Futuristic Starting Gun That Eliminates Advantages in Races the fully electronic, futuristic starting gun that eliminates advantages in races the fully electronic, futuristic starting gun that eliminates advantages in
    races"",""body"":""And that can be carried on a plane without the hassle too The Omega..."\t...\t"9"\t"0.181818182"\t"0.125448029"\t"1"']
```

显然，和原数据集一致，表明我们的数据加载无误（由于中间内容太多，…代表省略内容）。从以上数据格式可知，字段间由制表符 (\t) 进行分割。

同时，我们也可以使用 raw_data.first() 来查看第一行数据，显示内容与 raw_data.take(1) 是一样的内容，这里不再给出，读者可自行尝试。

3）查看数据文件的总行数：

```
In [3]: numRaws = raw_data.count()
In [4]: numRaws
Out[4]: 7395
```

4）按键进行统计：

```
In [5]: raw_data.countByKey()
Out[5]: defaultdict(int, {u'"': 7395})
```

原数据文件总的行数为 7396，由于我们已经在数据加载中去掉数据集的第一行数据，所以这里结果为 7395。



由于是 IPython 交互式环境，所以 shell 的指令在 PySpark 中依旧使用，如 clear 等。

11.5　数据预处理

1）由于后续的算法我们不需要时间戳以及网页的内容，所以这里先将其过滤掉。

```
In [6]: records = raw_data.map(lambda line: line.split('\t'))
```

2）查看 records 数据结构：

```
In [7]: records.first()
Out[7]:
[u'"http://www.bloomberg.com/news/2010-12-23/ibm-predicts-holographic-calls-air-breathing-batteries-by-2015.html"',
 u'"4042"',
 u'"{""title"":""IBM Sees Holographic Calls Air Breathing Batteries ibm sees holographic calls, air-breathing batteries"",""body"":""A sign stands outside the International Business Machines Corp IBM Almaden Research Center campus in San Jose California Photographer Tony Avelar..""}"',
 u'"business"',
 u'"0.789131"',
 u'"2.055555556"',
 u'"0.676470588"',
 u'"0.205882353"',
 u'"0.047058824"',
 u'"0.023529412"',
 u'"0.443783175"',
 u'"0"',
 u'"0"',
 u'"0.09077381"',
 u'"0"',
 u'"0.245831182"',
 u'"0.003883495"',
 u'"1"',
 u'"1"',
 u'"24"',
 u'"0"',
 u'"5424"',
 u'"170"',
 u'"8"',
 u'"0.152941176"',
 u'"0.079129575"',
 u'"0"']
```

3）查看每一行的列数：

```
In [8]: len(records.first())
Out[8]: 27
```

接下来，需要对数据进行清理工作，具体如下：

①去掉引号

②把标签列（即最后一列）转换为整数

③把第 4 列中的？转换为 0.0

导入 Vectors 矢量方法：

```
In [9]: from pyspark.ml.linalg import Vectors
```

导入决策树分类器：

```
In [10]: from pyspark.ml.classification import DecisionTreeClassifier
```

4）将 RDD 中的所有元素以列表的形式返回：

```
In [11]: data = records.collect()
```

5）查看 data 数据一行有多少列：

```
In [12]: numColumns = len(data[0])
In [13]: numColumns
Out[13]: 27
```

6）定义一个列表 data1，存放清理过的数据，格式为 [(label_1, features_1), (label_2, features_2),…]：

```
In [14]: data1 = []
```

对数据进行清理工作中的①②③步：

```
In [15]:
for i in range(numRaws):
    trimmed = [ each.replace('"', "") for each in data[i] ]
    label = int(trimmed[-1])
    features = map(lambda x: 0.0 if x == "?" else x, trimmed[4:numColumns-1])
    c = (label, Vectors.dense(map(float, features)))
    data1.append(c)
```

11.6　创建决策树模型

1）将 data1 转换为 DataFrame 对象，label 表示标签列，features 表示特征值列：

```
In [16]: df= spark.createDataFrame(data1, ["label","features"])
In [17]: df.show(10)
+-----+--------------------+
|label|            features|
+-----+--------------------+
|    0|[0.789131,2.05555...|
|    1|[0.574147,3.67796...|
|    1|[0.996526,2.38288...|
|    1|[0.801248,1.54310...|
|    0|[0.719157,2.67647...|
|    0|[0.0,119.0,0.7454...|
|    1|[0.22111,0.773809...|
|    0|[0.0,1.883333333,...|
|    1|[0.0,0.471502591,...|
|    1|[0.0,2.41011236,0...|
+-----+--------------------+
```

```
only showing top 10 rows

# 显示 df 的 Schema
In [18]: df.printSchema()
root
  |-- label: long (nullable = true)
  |-- features: vector (nullable = true)
```

2）由于后面会经常使用，所以将 df 载入内存：

```
In [19]: df.cache()
Out[19]: DataFrame[label: double, features: vector]
```

3）建立特征索引：

```
In [20]: from pyspark.ml.feature import VectorIndexer
In [20]: featureIndexer = VectorIndexer(inputCol="features",
 outputCol="indexedFeatures", maxCategories=24).fit(df)
```

4）将数据切分成 80% 训练集和 20% 测试集：

```
#seed=1234L, 表示每次随机生成的训练集和测试集的总行数不变
In [21]: (trainingData, testData) = df.randomSplit([0.8, 0.2],seed=1234L)

In [22]: trainingData.count()
Out[22]: 5912

In [23]: testData.count()
Out[23]: 1483
```

5）指定决策树模型的深度、标签列，特征值列，使用信息熵（entropy）作为评估方法，并训练数据。

```
In [24]: dt = DecisionTreeClassifier(maxDepth=5, labelCol="label",
featuresCol="indexedFeatures", impurity="entropy")
```

6）构建流水线工作流：

```
In [25]: from pyspark.ml import Pipeline

In [26]: pipeline = Pipeline(stages=[featureIndexer, dt])

In [27]: model = pipeline.fit(trainingData)        ## 训练模型
```

下一节我们用一组已知数据和一组新数据重新预测下结果。

11.7 训练模型进行预测

1）使用第一行数据进行预测结果，看看是否相符合，这里先来看一下原数据集的第一行

数据：

```
In [28]: data1[0]
Out[28]:
(0.0,
 DenseVector([0.7891, 2.0556, 0.6765, 0.2059, 0.0471, 0.0235, 0.4438, 0.0,
0.0, 0.0908, 0.0, 0.2458, 0.0039, 1.0, 1.0, 24.0, 0.0, 5424.0, 170.0, 8.0, 0.1529,
0.0791]))
```

2）使用数据集中第一行的特征值数据进行预测：

```
In [29]: test0 = spark.createDataFrame([(data1[0][1],)], ["features"])
In [30]: result = model.transform(test0)
# 查看预测结果
In [31]: result.show()
+--------------------+--------------------+-------------+--------------------+----------+
|            features|     indexedFeatures|rawPrediction|         probability|prediction|
+--------------------+--------------------+-------------+--------------------+----------+
|[0.789131,2.05555...|[0.789131,2.05555...|[274.0,310.0]|[0.46917808219178...|       1.0|
+--------------------+--------------------+-------------+--------------------+----------+
In [32]: predictedResult.select(['prediction']).show()  # 只获取预测值
+----------+
|prediction|
+----------+
|       1.0|
+----------+
```

3）将第一行的特征值数据修改掉 2 个（这里换掉第一个和第二个值），进行该特征值下的预测：

```
# 将第一行的数据进行修改
In [33]: firstRaw = list(data1[0][1])
In [34]: firstRaw[0] = 2.7891
In [35]: firstRaw[1] = 0.0556

In [36]: predictedData = Vectors.dense(firstRaw)
In [37]: predictedData
Out[37]: DenseVector([2.7891, 0.0556, 0.6765, 0.2059, 0.0471, 0.0235, 0.4438,
0.0, 0.0, 0.0908, 0.0, 0.2458, 0.0039, 1.0, 1.0, 24.0, 0.0, 5424.0, 170.0, 8.0,
0.1529, 0.0791])
```

4）进行新数据的预测：

```
In [38]: predictedRaw = spark.createDataFrame([(predictedData,)], ["features"])
In [39]: predictedResult = model.transform(predictedRaw)
In [40]: predictedResult.show()
+--------------------+--------------------+-------------+--------------------+----------+
|            features|     indexedFeatures|rawPrediction|         probability|prediction|
+--------------------+--------------------+-------------+--------------------+----------+
|[2.7891,0.0556,0....|[2.7891,0.0556,0....|[274.0,310.0]|[0.46917808219178...|       1.0|
+--------------------+--------------------+-------------+--------------------+----------+
```

```
In [41]: predictedResult.select(['prediction']).show()
+----------+
|prediction|
+----------+
|       1.0|
+----------+
```

5）下面我们用测试数据做决策树准确度测试：

```
# 通过模型，预测测试集
In [42]: predictedResultAll = model.transform(testData)

# 查看预测值
In [43]: predictedResultAll.select("prediction").show()
+----------+
|prediction|
+----------+
|       0.0|
|       0.0|
|       1.0|
|       1.0|
|       0.0|
|       0.0|
|       0.0|
|       0.0|
|       0.0|
|       1.0|
+----------+
only showing top 10 rows

# 由于预测值是 DataFrame 对象，每一行是 Raw 型，不可做修改
# 需将预测值转换为 pandas，然后转换为列表
In [44]:df_prediction = predictedResultAll.select("prediction").toPandas()
In [45]: dtPredictions = list(df_prediction.prediction)

# 查看前 10 个预测值
In [46]: dtPredictions[:10]
Out[46]: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0]

# 对预测值做准确性统计
In [47]: dtTotalCorrect = 0
# 获取测试集的总行数
In [48]: testRaw = testData.count()
In [49]: testLabel = testData.select("label").collect()
In [50]:
for i in range(testRaw):
    if dtPredictions[i] == testLabel[i]:
        dtTotalCorrect += 1

In [51]: dtTotalCorrect
Out[51]: 940
```

```
In [52]: 1.0 * dtTotalCorrect / testRaw
Out[52]: 0.6338503034389751
```

11.8　模型优化

在上一个小节中，我们发现使用决策树的正确率不算高，只有 63.3850%。在这一小节，我们探究一下改进预测准确率的方法。

提到优化，一般会从以下方面考虑：

- 数据的覆盖面是否恰当？
- 数据是否清理过（如去噪等）？
- 数据是否规范（标准化、正则化）？
- 数据的维度是否合适？
- 类别是否处理？
- 参数、算法是否最好？
- 如果出现过拟合，是否采用了交叉测试等方法？
- 是否还有其他方法。

11.8.1　特征值的优化

由于对原数据只做了简单的清理，而且数据取得也不完全，只取了第 5 列到第 27 列。查看原数据可以发现，第 4 列数据是标识该网页的性质的，依照人们的正常的想法，这往往是评判一个网页是长久的还是短暂类型的一个很重要的参数。

1）先将之前用到的一些代码加载进来。

```
In [1]: from pyspark.ml.linalg import Vectors
In [2]: from pyspark.ml.classification import DecisionTreeClassifier
In [3]: from pyspark.ml.feature import VectorIndexer
In [4]: from pyspark.ml import Pipeline
In [5]: raw_data = sc.textFile("hdfs://master:9000/data/train_noheader.tsv")
In [6]: numRaws = raw_data.count()
In [7]: records = raw_data.map(lambda line: line.split('\t'))
In [8]: data = records.collect()
In [9]: numColumns = len(data[0])
In [10]: data1 = []
```

2）由于这里对网页类型的标识有很多，需要单独挑选出来进行处理。

```
# 将第三列网页类型的引号去除掉
In [11]: category = records.map(lambda x: x[3].replace("\"",""))
```

将网页的唯一类型筛选出来，并进行排序。

```
In [12]: categories = sorted(category.distinct().collect())
In [13]: categories
Out[13]:
[u'?',
 u'arts_entertainment',
 u'business',
 u'computer_internet',
 u'culture_politics',
 u'gaming',
 u'health',
 u'law_crime',
 u'recreation',
 u'religion',
 u'science_technology',
 u'sports',
 u'unknown',
 u'weather']
```

3）查看网页类型的个数。

```
In [14]: numCategories = len(categories)
In [15]: numCategories
Out[15]: 14
```

我们就可以使用“One of K”来标签进行转换，即 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] 这样一个列表。如果网页标识为“?”，那么将其对应列表的值置为 1，即 [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]；同理，如果网页性质标识为“wether”，那相应的列表为 [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]。

4）紧接着，我们定义一个函数，用于返回当前网页类型的列表。

```
In [16]:
def transform_category(x):
    markCategory = [0] * numCategories
    index = categories.index(x)
    markCategory[index] = 1
    return markCategory
```

5）通过这样的处理，我们将网页类型这一特征值转化 14 个特征值，整体的特征值其实就增加了 14 个。接下来，我们在处理的时候将这些特征值加入进去。

```
In [17]:
for i in range(numRaws):
    trimmed = [ each.replace('"', "") for each in data[i] ]
    label = float(trimmed[-1])
    cate = transform_category(trimmed[3]) # 调用函数，返回一个类型列表
    features = cate + map(lambda x: 0.0 if x == "?" else (x), trimmed[4:numColumns -
1])
    c = (label, Vectors.dense(map(float, features)))
    data1.append(c)
```

6）创建 DataFrame 对象。

```
In [18]: df= spark.createDataFrame(data1, ["label","features"])

# 由于后面经常使用 df，所以载入内存
In [19]: df.cache()
Out[20]: DataFrame[label: double, features: vector]
```

7）建立特征索引。

```
In [21]: from pyspark.ml.feature import VectorIndexer
In [22]: featureIndexer = VectorIndexer(inputCol="features",
outputCol="indexedFeatures", maxCategories=24).fit(df)
```

8）将数据切分成 80% 训练集和 20% 测试集。

```
In [23]:(trainingData, testData) = df.randomSplit([0.8, 0.2],seed=1234L)

In [24]: trainingData.count()
Out[24]: 5912

In [25]: testData.count()
Out[25]: 1483
```

9）创建决策树模型。

```
In [26]: dt = DecisionTreeClassifier(maxDepth=5, labelCol="label",
featuresCol="indexedFeatures", impurity="entropy")
```

10）构建流水线工作流。

```
In [27]: pipeline = Pipeline(stages=[featureIndexer, dt])
In [28]: model = pipeline.fit(trainingData)        ## 训练模型
```

11）用测试数据再一次做下决策树准确度测试。

```
In [29]: predictedResultAll = model.transform(testData)
In [30]:df_prediction = predictedResultAll.select("prediction").toPandas()
In [31]: dtPredictions = list(df_prediction.prediction)

# 对预测值做准确性统计
In [32]: dtTotalCorrect = 0
# 测试集的总行数
In [33]: testRaw = testData.count()
In [49]: testLabel = testData.select("label").collect()
In [34]:
for i in range(testRaw):
    if dtPredictions[i] == testLabel[i]:
        dtTotalCorrect += 1

In [35]: dtTotalCorrect
Out[35]: 967
```

```
In [36]: 1.0 * dtTotalCorrect / testRaw
Out[36]: 0.6520566419420094
```

可以看到，准确率增大到了 65.2057%，而未做优化前的准确率是 63.3850%。增长了 1.88%。效果还是比较显著的。

11.8.2 交叉验证和网格参数

上一小节，我们从特征值的方面进行了模型的优化，其实模型的优化工作还有很多，例如从准确率和召回率入手，对模型进行进一步优化。在这一小节中，我们将使用交叉验证和网格参数的手段，对模型进行优化。

关于交叉验证和网格参数的原理，这里不再赘述，在本书第 5 章模型选择和优化一章中，已经做了较为详细的说明。下面我们在上一小节模型优化的基础上，用决策树的最大深度使用参数网格和交叉验证的手段对模型进行进一步的优化，通过程序自动选择决策树的最大深度来获取准确率最高的模型。

```
In [1]: from pyspark.ml.linalg import Vectors
In [2]: from pyspark.ml.classification import DecisionTreeClassifier
In [3]: from pyspark.ml.feature import VectorIndexer
In [4]: from pyspark.ml import Pipeline
In [5]: raw_data = sc.textFile("hdfs://master:9000/data/train_noheader.tsv")
In [6]: numRaws = raw_data.count()
In [7]: records = raw_data.map(lambda line: line.split('\t'))
In [8]: data = records.collect()
In [9]: numColumns = len(data[0])
In [10]: data1 = []
In [11]: category = records.map(lambda x: x[3].replace("\"",""))
In [12]: categories = sorted(category.distinct().collect())
In [13]: numCategories = len(categories)
In [14]:
def transform_category(x):
    markCategory = [0] * numCategories
    index = categories.index(x)
    markCategory[index] = 1
    return markCategory
In [15]:
for i in range(numRaws):
    trimmed = [ each.replace('"', "") for each in data[i] ]
    label = float(trimmed[-1])
    cate = transform_category(trimmed[3]) # 调用函数，返回一个类型列表
    features = cate + map(lambda x: 0.0 if x == "?" else (x), trimmed[4:numColumns - 1])
    c = (label, Vectors.dense(map(float, features)))
    data1.append(c)
In [16]: df= spark.createDataFrame(data1, ["label","features"])
In [17]: df.cache()
In [18]: featureIndexer = VectorIndexer(inputCol="features", outputCol="indexedFeatures",
```

```
maxCategories=24).fit(df)
    In [19]: dt = DecisionTreeClassifier(maxDepth=5, labelCol="label", featuresCol="
indexedFeatures", impurity="entropy")
    In [20]: pipeline = Pipeline(stages=[featureIndexer, dt])
    In [21]: (trainingData, testData) = df.randomSplit([0.8, 0.2],seed=1234L)
```

创建交叉验证和网格参数：

```
    # 导入交叉验证和参数网格
    In [22]: from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
    # 导入二分类评估器
    In [23]: from pyspark.ml.evaluation import BinaryClassificationEvaluator
    In [24]: evaluator = BinaryClassificationEvaluator()  # 初始化一个评估器
    # 设置参数网格
    In [25]: paramGrid = ParamGridBuilder().addGrid(dt.maxDepth, [4,5,6]).build()
    # 设置交叉验证的参数
    In [26]: cv = CrossValidator().setEstimator(pipeline).setEvaluator(evaluator).se
tEstimatorParamMaps(paramGrid).setNumFolds(2)
```

通过交叉验证来训练模型：

```
    In [27]: cvModel = cv.fit(trainingData)
```

测试模型：

```
    In [28]: Predictions=cvModel.transform(testData)
```

准确率统计：

```
    In [29]: df_prediction = Predictions.select("prediction").toPandas()
    In [30]: dtPredictions = list(df_prediction.prediction)

    # 对预测值做准确性统计
    In [31]: dtTotalCorrect = 0
    # 测试集的总行数
    In [32]: testRaw = testData.count()
    In [34]: testLabel = testData.select("label").collect()
    In [33]:
    for i in range(testRaw):
        if dtPredictions[i] == testLabel[i]:
            dtTotalCorrect += 1

    In [34]: dtTotalCorrect
    Out[34]: 960

    In [35]: 1.0 * dtTotalCorrect / testRaw
    Out[35]: 0.6473364801078895
```

我们还可以查看最匹配模型的具体参数：

```
    In [36]: bestmodel = cvModel.bestModel.stages[1]
```

```
In [37]: bestmodel.numFeatures    # 决策树有 36 个特征值
Out[37]: 36

In [38]: bestmodel.depth  # 最大深度为 10
Out[38]: 6

In [39]: bestmodel.numNodes  # 决策树中点有 457 个
```

11.9 脚本方式运行

11.9.1 在脚本中添加配置信息

创建一个 decisionTree.py 文件，添加如下代码来配置启动 PySpark。将上述在 PySpark 的 IPython 中的代码添加到该文件中来。

本文的示例程序存为 /home/hadoop/projects/spark/pyspark/decisionTree.py。

```
from pyspark import SparkConf, SparkContext
from pyspark.sql import SparkSession

# 指定本地运行 spark
conf = SparkConf().setMaster("local[*]")
sc = SparkContext(conf=conf)
spark = SparkSession.builder.master('local') \
                .appName('DecisionTree') \
                .config("spark.some.config.option", "some-value") \
                .getOrCreate()
```

11.9.2 运行脚本程序

spark 2.0 之前使用 pyspark decisionTree.py 来执行文件，spark 2.0 之后统一用 spark-submit decisionTree.py 执行文件。读者可以使用 spark-submit --help 来查看相关命令的帮助信息。

```
$ spark-submit decisionTree.py
```

11.10 小结

本章我们了解 Spark 中的 PySpark 使用方法，对于 PySpark 做了简单的介绍。讨论了分类模型中最常见的决策树模型在 PySpark 的应用，用实例讲解了如何对数据进行清理、转换，分析了分类模型的准确性有待提高的原因，通过可视化对决策树在不同深度下的准确度进行讨论。

第 12 章 Chapter 12

SparkR 朴素贝叶斯模型

前一章我们介绍了 PySpark，就是用 Python 语言操作 Spark 大数据计算框架上的任务，这样就自然把 Python 的优点与 Spark 的优势进行叠加。Spark 提供了 Python 的 API，也提供了 R 语言的 API，其组件名称为 SparkR。SparkR 的运行原理或架构如图 12-1 所示。

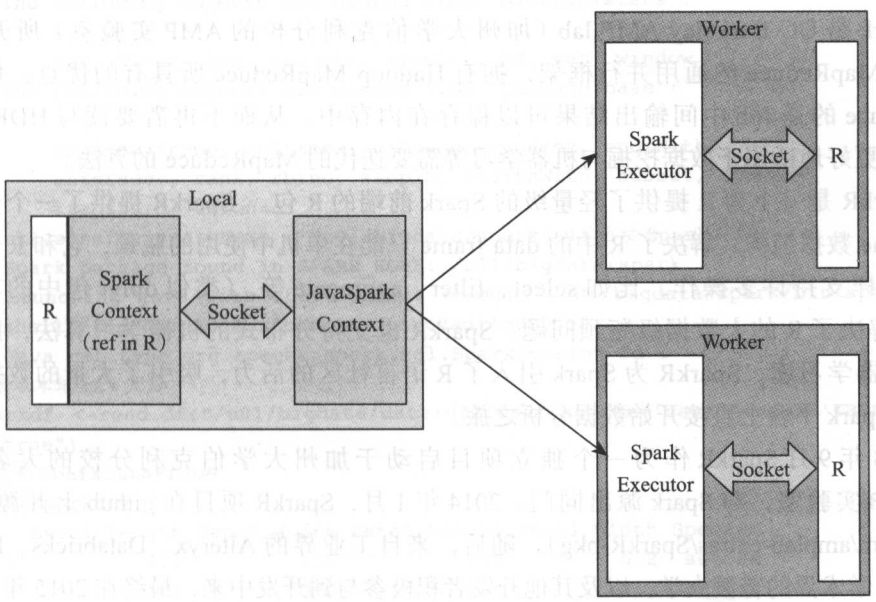


图 12-1 SparkR 架构图

SparkR 的架构类似于 PySpark，Driver 端除了一个 JVM 进程外（包含一个 SparkContext，

在 Spark2.X 中 SparkContext 已经被 SparkSession 所代替），还有一个 R 的进程，这两个进程通过 Socket 进行通信，用户可以提交 R 语言代码，R 的进程会执行这些 R 代码，当 R 代码调用 Spark 相关函数时，R 进程会通过 Socket 触发 JVM 中的对应任务。

当 R 进程向 JVM 进程提交任务的时候，R 会把子任务需要的环境进行打包，并发送到 JVM 的 driver 端。通过 R 生成的 RDD 都会是 RRDD 类型，当触发 RRDD 的 action 时，Spark 的执行器会开启一个 R 进程，执行器和 R 进程通过 Socket 进行通信。执行器会把任务和所需的环境发送给 R 进程，R 进程会加载对应的 package，执行任务，并返回结果。

本章通过一个实例来说明如何使用 SparkR，具体内容如下：

- ❑ SparkR 简介；
- ❑ 把数据上传到 HDFS，然后导入 Hive，最后从 Hive 读取数据；
- ❑ 使用朴素贝叶斯分类器；
- ❑ 探索数据；
- ❑ 预处理数据；
- ❑ 训练模型；
- ❑ 评估模型；

12.1 SparkR 简介

Spark 是 UC Berkeley AMP lab（加州大学伯克利分校的 AMP 实验室）所开源的类 Hadoop MapReduce 的通用并行框架，拥有 Hadoop MapReduce 所具有的优点；但不同于 MapReduce 的是 Job 中间输出结果可以保存在内存中，从而不再需要读写 HDFS，因此 Spark 能更好地适用于数据挖掘与机器学习等需要迭代的 MapReduce 的算法。

SparkR 是一个为 R 提供了轻量级的 Spark 前端的 R 包。SparkR 提供了一个分布式的 data frame 数据结构，解决了 R 中的 data frame 只能在单机中使用的瓶颈，它和 R 中的 data frame 一样支持许多操作，比如 select、filter、aggregate 等。（类似 dplyr 包中的功能）这很好地解决了 R 的大数据级瓶颈问题。SparkR 也支持分布式的机器学习算法，比如使用 MLib 机器学习库。SparkR 为 Spark 引入了 R 语言社区的活力，吸引了大量的数据科学家开始在 Spark 平台上直接开始数据分析之旅。

2013 年 9 月 SparkR 作为一个独立项目启动于加州大学伯克利分校的大名鼎鼎的 AMPLAB 实验室，与 Spark 源出同门。2014 年 1 月，SparkR 项目在 github 上开源（https://github.com/amplab-extras/SparkR-pkg）。随后，来自工业界的 Alteryx、Databricks、Intel 等公司和来自学术界的普渡大学，以及其他开发者积极参与到开发中来，最终在 2015 年 4 月成功地合并进 Spark 代码库的主干分支，并在 Spark1.4 版本中作为重要的新特性之一正式宣布。

目前 SparkR 的最新版本为 2.0.1，API 参考文档：http://spark.apache.org/docs/latest/api/R/index.html。

12.2　获取数据

12.2.1　SparkDataFrame 数据结构说明

SparkDataFrame 是 Spark 提供的分布式数据格式（DataFrame）。类似于关系数据库中的表或 R 语言中的 DataFrame。SparkDataFrames 可以从各种各样的源构造，例如，结构化数据文件、Hive 中的表、外部数据库或现有的本地数据。

12.2.2　创建 SparkDataFrame

1. 从本地文件加载数据，生成 SparkDataFrame

SparkR 支持通过 SparkDataFrame 接口对各种数据源进行操作。从数据源创建 SparkDataFrame 的函数是 read.df。SparkR 支持本地读取 JSON、CSV 和 Parquet 文件；并且通过第三方项目的软件包，你可以找到像 Avro 这样的流行文件格式的数据源连接器。可以通过使用 spark-submit 或 SparkR 命令指定 packages 或者在交互式 R shell 或 RStudio 中使用 sparkPackages 参数初始化 SparkSession 来添加这些包。示例：

```
library("SparkR", lib.loc="/u01/bigdata/spark/R/lib")# 加载 sparkR
##
## Attaching package: 'SparkR'
## The following objects are masked from 'package:stats':
##
##     cov, filter, lag, na.omit, predict, sd, var, window
## The following objects are masked from 'package:base':
##
##     as.data.frame, colnames, colnames<-, drop, endsWith,
##     intersect, rank, rbind, sample, startsWith, subset, summary,
##     transform, union
sparkR.session(sparkHome ='/u01/bigdata/spark' )# 启动 Spark 环境
## Spark package found in SPARK_HOME: /u01/bigdata/spark
## Launching java with spark-submit command /u01/bigdata/spark/bin/spark-submit
sparkr-shell /tmp/RtmpA30Gvz/backend_port2ad11c0705d8
## Java ref type org.apache.spark.sql.SparkSession id 1
# 读取本地 csv 文件
Sparkdf <-read.df("/u01/bigdata/data/df2.csv",source='csv',header='TRUE',inferSc
hema ="true")
# 查看 SparkDataFrame
head(Sparkdf)
##   Sepal_Length Sepal_Width Petal_Length Petal_Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

2. 利用 R 环境的 data frames 创建 SparkDataFrame

创建 SparkDataFrame 的最简单的方法是将本地 R 环境变量中的 data frames 转换为 SparkDataFrame。我们可以使用 as.DataFrame 或 createDataFrame 函数来创建 SparkDataFrame。作为示例，我们使用 R 自带的 iris 数据集来创建 SparkDataFrame。

```
library("SparkR", lib.loc="/u01/bigdata/spark/R/lib")# 加载 sparkR
##
## Attaching package: 'SparkR'
## The following objects are masked from 'package:stats':
##
##     cov, filter, lag, na.omit, predict, sd, var, window
## The following objects are masked from 'package:base':
##
##     as.data.frame, colnames, colnames<-, drop, endsWith,
##     intersect, rank, rbind, sample, startsWith, subset, summary,
##     transform, union
sparkR.session(sparkHome ='/u01/bigdata/spark' )# 启动 Spark 环境
## Java ref type org.apache.spark.sql.SparkSession id 1
# 创建 SparkDataFrame Sparkdf，数据来自 iris 数据集
Sparkdf <-as.DataFrame(iris)
# 查看刚创建好的 SparkDataFrame
head(Sparkdf)
##   Sepal_Length Sepal_Width Petal_Length Petal_Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

3. 从 HDFS 文件系统加载数据，生成 SparkDataFrame

示例：

```
library("SparkR", lib.loc="/u01/bigdata/spark/R/lib")# 加载 sparkR
##
## Attaching package: 'SparkR'
## The following objects are masked from 'package:stats':
##
##     cov, filter, lag, na.omit, predict, sd, var, window
## The following objects are masked from 'package:base':
##
##     as.data.frame, colnames, colnames<-, drop, endsWith,
##     intersect, rank, rbind, sample, startsWith, subset, summary,
##     transform, union
```

```
## Java ref type org.apache.spark.sql.SparkSession id 1
# 读取 HDFS 文件
Sparkdf <-read.df("hdfs://192.168.1.112:9000/u01/bigdata/data/df2.csv",source='c
sv',header='TRUE',inferSchema ="true")
# 查看 SparkDataFrame
head(Sparkdf)
##   Sepal_Length Sepal_Width Petal_Length Petal_Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

4. 读取 Hive 数据仓库中的表，生成 SparkDataFrame

我们还可以从 Hive 表创建 SparkDataFrame。要做到这一点，我们需要创建一个具有 Hive 支持的 SparkSession，它可以访问 Hive MetaStore 中的表。**请注意，Spark 安装时应该选择支持 Hive 方式进行安装。**在 SparkR 中，默认情况下，它将尝试创建一个启用了 Hive 支持的 SparkSession（enableHiveSupport = TRUE）。

```
library("SparkR", lib.loc="/u01/bigdata/spark/R/lib")# 加载 sparkR
# 查看数据库
sql('show databases')
## SparkDataFrame[databaseName:string]
# 选择 hive 库
sql('use hive')
## SparkDataFrame[]
# 查看 hive 数据库的表
sql('show tables')
## SparkDataFrame[database:string, tableName:string, isTemporary:boolean]
# 查看表 df2 的信息
sql('desc df2')
## SparkDataFrame[col_name:string, data_type:string, comment:string]
# 读取 hive 表 df2，生成 SparkDataFrame
Sparkdf<-sql('select * from df2')
# 查看 SparkDataFrame
head(Sparkdf)
##       height     weight
## 1  0.3307575 -1.4197984
## 2  0.4970992 -1.4364733
## 3  1.4477968 -0.7579736
## 4  0.6815300 -1.7573564
## 5  0.8915567  1.1815332
## 6 -2.2494993 -1.6438995
```

12.2.3 SparkDataFrame 的常用操作

1. 选择行，或者列

```
df <-as.DataFrame(iris)
str(df)
## 'SparkDataFrame': 5 variables:
##  $ Sepal_Length: num 5.1 4.9 4.7 4.6 5 5.4
##  $ Sepal_Width : num 3.5 3 3.2 3.1 3.6 3.9
##  $ Petal_Length: num 1.4 1.4 1.3 1.5 1.4 1.7
##  $ Petal_Width : num 0.2 0.2 0.2 0.2 0.2 0.4
##  $ Species     : chr "setosa" "setosa" "setosa" "setosa" "setosa" "setosa"
head(df)
##   Sepal_Length Sepal_Width Petal_Length Petal_Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
# 选择 Sepal_Length 列
head(select(df, df$Sepal_Length))
##   Sepal_Length
## 1          5.1
## 2          4.9
## 3          4.7
## 4          4.6
## 5          5.0
## 6          5.4
# 或者
head(select(df, "Sepal_Length"))
##   Sepal_Length
## 1          5.1
## 2          4.9
## 3          4.7
## 4          4.6
## 5          5.0
## 6          5.4
# 过滤出 Sepal_Length 小于 5 的行
head(filter(df, df$Sepal_Length <5))
##   Sepal_Length Sepal_Width Petal_Length Petal_Width Species
## 1          4.9         3.0          1.4         0.2  setosa
## 2          4.7         3.2          1.3         0.2  setosa
## 3          4.6         3.1          1.5         0.2  setosa
## 4          4.6         3.4          1.4         0.3  setosa
## 5          4.4         2.9          1.4         0.2  setosa
## 6          4.9         3.1          1.5         0.1  setosa
```

2. 数据分组，聚合

```
df <-as.DataFrame(faithful)
# 数据分组，并统计每组出现的个数
head(summarize(groupBy(df, df$waiting), count =n(df$waiting)))
##   waiting count
## 1      70     4
## 2      67     1
## 3      69     2
## 4      88     6
## 5      49     5
## 6      64     4
# 对结果进行排序
waiting_counts <-summarize(groupBy(df, df$waiting), count =n(df$waiting))
head(arrange(waiting_counts, desc(waiting_counts$count)))
##   waiting count
## 1      78    15
## 2      83    14
## 3      81    13
## 4      77    12
## 5      82    12
## 6      79    10
```

3. 对 SparkDataFrame 的列进行运算操作

```
df$waiting_secs <-df$waiting *60
head(df)
##   eruptions waiting waiting_secs
## 1     3.600      79         4740
## 2     1.800      54         3240
## 3     3.333      74         4440
## 4     2.283      62         3720
## 5     4.533      85         5100
## 6     2.883      55         3300
```

4. apply 系列函数的应用

```
dapply 函数类似于 R 语言的 apply 函数，看一个示例。
df <-as.DataFrame(iris)
df1 <-dapply(df, function(x) { x[x[,1]>6,]},schema =schema(df))
head(collect(df1))
##   Sepal_Length Sepal_Width Petal_Length Petal_Width    Species
## 1          7.0         3.2          4.7         1.4 versicolor
## 2          6.4         3.2          4.5         1.5 versicolor
## 3          6.9         3.1          4.9         1.5 versicolor
## 4          6.5         2.8          4.6         1.5 versicolor
```

```
## 5          6.3          3.3          4.7          1.6 versicolor
## 6          6.6          2.9          4.6          1.3 versicolor
str(df1)
## 'SparkDataFrame': 5 variables:
##  $ Sepal_Length: num 7 6.4 6.9 6.5 6.3 6.6
##  $ Sepal_Width : num 3.2 3.2 3.1 2.8 3.3 2.9
##  $ Petal_Length: num 4.7 4.5 4.9 4.6 4.7 4.6
##  $ Petal_Width : num 1.4 1.5 1.5 1.5 1.6 1.3
##  $ Species     : chr "versicolor" "versicolor" "versicolor" "versicolor"
"versicolor" "versicolor"
dim(df1)
## [1] 61  5
```

其他函数（如 dapplyCollect 函数、gapply 函数等）的使用方法，请自行参考相关资料，这里不再赘述。

12.3 朴素贝叶斯分类器

该案例数据来自泰坦尼克号人员存活情况，响应变量为 Survived，包含 2 个分类（yes，no），特征变量有 Sex、Age、Class（船舱等级），说明如下：

Class :0 = crew, 1 = first, 2 = second, 3 = third Age :1 = adult, 0 = child Sex :1 = male, 0 = female Survived :1 = yes, 0 = no

12.3.1 数据探查

让我们来观察一下数据。

```
library("SparkR", lib.loc="/u01/bigdata/spark/R/lib")# 加载 sparkR
##
## Attaching package: 'SparkR'
## The following objects are masked from 'package:stats':
##
##     cov, filter, lag, na.omit, predict, sd, var, window
## The following objects are masked from 'package:base':
##
##     as.data.frame, colnames, colnames<-, drop, endsWith,
##     intersect, rank, rbind, sample, startsWith, subset, summary,
##     transform, union
## Java ref type org.apache.spark.sql.SparkSession id 1
## SparkDataFrame[]
# 从 hive 仓库加载数据
titanic <-sql('select * from titanic')
# 查看 SparkDataFrame
head(titanic)
```

```
##   class age sex survived
## 1     1   1   1        1
## 2     1   1   1        1
## 3     1   1   1        1
## 4     1   1   1        1
## 5     1   1   1        1
## 6     1   1   1        1
dim(titanic)
## [1] 2201    4
# 查看 SparkDataFrame
dim(titanic)# 查看数据的记录数以及维度数量
## [1] 2201    4
```

12.3.2　对原始数据集进行转换

```
titanic_df=as.data.frame(titanic)
titanic <-as.data.frame(table(titanic_df))
colnames(titanic)<-paste0(toupper(substring(colnames(titanic),1,1)),substring(co
lnames(titanic),2))
titanic_temp<-titanic[titanic$Freq >0, -5]
head(titanic_temp)
##    Class Age Sex Survived
## 4      3   0   0        0
## 5      0   1   0        0
## 6      1   1   0        0
## 7      2   1   0        0
## 8      3   1   0        0
## 12     3   0   1        0
```

12.3.3　查看不同船舱的生还率差异

```
library(ggplot2)
library(dplyr)
##
## Attaching package: 'dplyr'
## The following objects are masked from 'package:SparkR':
##
##     arrange, between, collect, contains, count, cume_dist,
##     dense_rank, desc, distinct, explain, filter, first, group_by,
##     intersect, lag, last, lead, mutate, n, n_distinct, ntile,
##     percent_rank, rename, row_number, sample_frac, select, sql,
##     summarize, union
## The following objects are masked from 'package:stats':
##
```

```
##     filter, lag
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
tempdata<-aggregate(Freq~Class+Survived,data = titanic,FUN = sum)
ggplot(data = tempdata,mapping =aes(x = Class,y=Freq,fill=Survived))+geom_
bar(position ='dodge',stat ='identity')+ylab("number")+xlim(c("1","2","3","0"))+them
e(text=element_text(family ="Italic",size=18))
```

由图 12-2 可以看出，头等舱的生还率最高，船员以及三等舱的生还率较低。

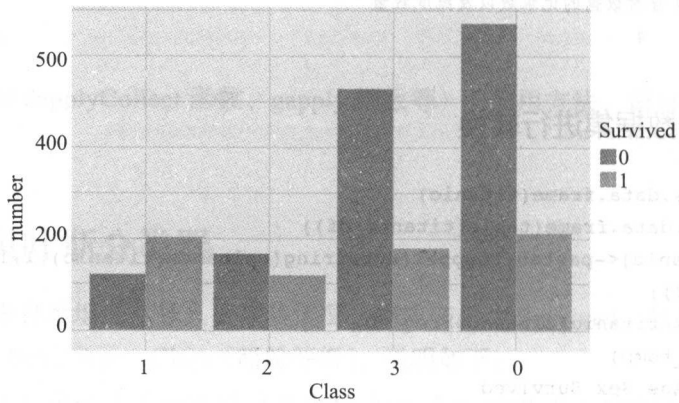


图 12-2 不同船舱的生还率差异

然后，对比一下不同性别之间的生还率：

```
tempdata<-aggregate(Freq~Sex+Survived,data = titanic,FUN = sum)
ggplot(data = tempdata,mapping =aes(x = Sex,y=Freq,fill=Survived))+geom_
bar(position ='dodge',stat ='identity')
```

由图 12-3 可知，女性生还率较高。

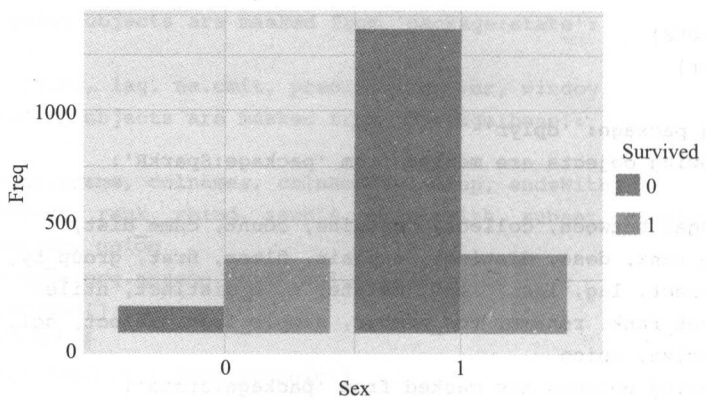


图 12-3 不同性别的生还率差异

最后再看看不同年龄段的生还情况：

```
tempdata<-aggregate(Freq~Age+Survived,data = titanic,FUN = sum)
ggplot(data = tempdata,mapping =aes(x = Age,y=Freq,fill=Survived))+geom_
bar(position ='dodge',stat ='identity')
```

如图 12-4 所示，儿童的生还率较高。

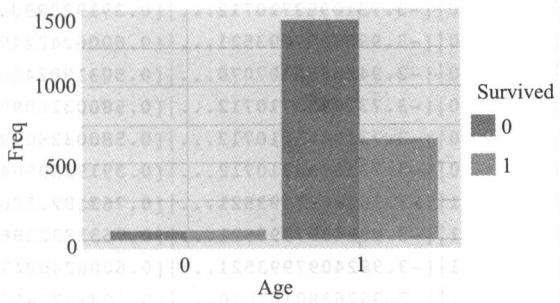


图 12-4　不同年龄的生还率差异

12.3.4　转换成 SparkDataFrame 格式的数据

```
titanicDF <-createDataFrame(titanic[titanic$Freq >0, -5])
nbDF <-titanicDF
nbTestDF <-titanicDF
# 使用 spark.naiveBayes 方法，训练模型
nbModel <-spark.naiveBayes(nbDF, Survived ~Class +Sex +Age)
```

12.3.5　模型概要

```
summary(nbModel)
## $apriori
##              1         0
## [1,] 0.5769231 0.4230769
##
## $tables
##   Class_3   Class_2 Class_1 Sex_0 Age_1
## 1 0.3125    0.3125  0.3125  0.5   0.5625
## 0 0.4166667 0.25    0.25    0.5   0.75
```

12.3.6　预测

```
nbPredictions <-predict(nbModel, nbTestDF)
showDF(nbPredictions)
```

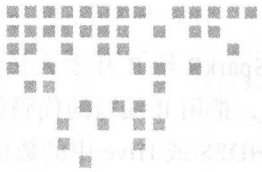
```
## +-----+---+---+--------+--------------------+--------------------+----------+
## |Class|Age|Sex|Survived|       rawPrediction|         probability|prediction|
## +-----+---+---+--------+--------------------+--------------------+----------+
## |    3|  0|  0|       0|[-3.9824097993521...|[0.60062402496099...|         1|
## |    0|  1|  0|       0|[-2.9426380107070...|[0.50318824507901...|         1|
## |    1|  1|  0|       0|[-3.7310953710712...|[0.58003280993672...|         1|
## |    2|  1|  0|       0|[-3.7310953710712...|[0.58003280993672...|         1|
## |    3|  1|  0|       0|[-3.7310953710712...|[0.39192399049881...|         0|
## |    3|  0|  1|       0|[-3.9824097993521...|[0.60062402496099...|         1|
## |    0|  1|  1|       0|[-2.9426380107070...|[0.50318824507901...|         1|
## |    1|  1|  1|       0|[-3.7310953710712...|[0.58003280993672...|         1|
## |    2|  1|  1|       0|[-3.7310953710712...|[0.58003280993672...|         1|
## |    3|  1|  1|       0|[-3.7310953710712...|[0.39192399049881...|         0|
## |    1|  0|  0|       1|[-3.9824097993521...|[0.76318223866790...|         1|
## |    2|  0|  0|       1|[-3.9824097993521...|[0.76318223866790...|         1|
## |    3|  0|  0|       1|[-3.9824097993521...|[0.60062402496099...|         1|
## |    0|  1|  0|       1|[-2.9426380107070...|[0.50318824507901...|         1|
## |    1|  1|  0|       1|[-3.7310953710712...|[0.58003280993672...|         1|
## |    2|  1|  0|       1|[-3.7310953710712...|[0.58003280993672...|         1|
## |    3|  1|  0|       1|[-3.7310953710712...|[0.39192399049881...|         0|
## |    1|  0|  1|       1|[-3.9824097993521...|[0.76318223866790...|         1|
## |    2|  0|  1|       1|[-3.9824097993521...|[0.76318223866790...|         1|
## |    3|  0|  1|       1|[-3.9824097993521...|[0.60062402496099...|         1|
## +-----+---+---+--------+--------------------+--------------------+----------+
## only showing top 20 rows
```

12.3.7 评估模型

```
nbPredictions<-as.data.frame(nbPredictions)
# 计算混淆矩阵
ct<-table(titanic_temp$Survived,nbPredictions$prediction)
ct
##
##      0  1
##   0  2  8
##   1  2 12
# 计算准确率
(ct[1,1]+ct[2,2])/sum(ct)
## [1] 0.5833333
# 计算召回率
ct[2,2]/(ct[2,2]+ct[2,1])
## [1] 0.8571429
# 计算精准率
ct[2,2]/(ct[2,2]+ct[1,2])
## [1] 0.6
```

12.4　小结

本章主要讲解了如何使用 SparkR 组件的问题，SparkR 给 R 开发人员提供很多 API，利用这些 API，开发人员就可以通过 R 语言操作 Spark，把用 R 编写的代码放在 Spark 这个大数据技术平台上运行，这样可以使 R 不但可以操作 HDFS 或 Hive 中的数据，而且可以使用 Spark 分布式基于内存的架构。



Chapter 13 第 13 章

使用 Spark Streaming 构建在线学习模型

前面我们介绍的这些算法，一般基于一个或几个相对固定的文件，以这样的数据为模型处理的源数据是固定的，这样的数据或许很大、很多。训练或测试都是建立在这些固定数据之上，当然，测试时，可能会取这个数据源之外的数据，如新数据或其他数据等。训练模型的数据一般是相对固定的，这样的机器学习的场景是很普遍的。

但实际环境中，还有其他一些场景，如源数据经常变换，就像流水一样，时刻在变换着，例如很多在线数据、很多日志数据等。面对这些数据的学习我们该如何处理呢？

这个问题实际上属于流水计算问题，目前解决这类问题主要使用 Spark Streaming、Storm、Samza 等。这章我们主要介绍 Spark Streaming。

本章主要包括以下内容：

- 介绍 Spark Streaming；
- Spark Streaming 入门实例；
- 在线学习实例。

13.1 Spark Streaming 简介

Spark Streaming 是 Spark 核心 API 的一个扩展，可以实现高吞吐量的、具备容错机制的实时流数据的处理。支持从多种数据源获取数据，包括 Kafk、Flume、Twitter、ZeroMQ、Kinesis 以及 TCP sockets，从数据源获取数据之后，可以使用诸如 map、reduce、join 和 window 等高级函数进行复杂算法的处理。最后还可以将处理结果存储到文件系统，数据库和现场仪表盘。在“One Stack rule them all”的基础上，还可以使用 Spark 的其他子框架，

如集群学习、图计算等，对流数据进行处理。

13.1.1　Spark Streaming 常用术语

在简介 Spark Streaming 前，我们先简单介绍 Streaming 的一些常用术语。

- **离散流（discretized stream）或 DStream**：这是 Spark Streaming 对内部持续的实时数据流的抽象描述，即我们处理的一个实时数据流，在 Spark Streaming 中对应于一个 DStream 实例。
- **批数据（batch data）**：这是化整为零的第一步，将实时流数据以时间片为单位进行分批，将流处理转化为时间片数据的批处理。随着持续时间的推移，这些处理结果就形成了对应的结果数据流了。
- **时间片或批处理时间间隔（batch interval）**：这是人为地对流数据进行定量的标准，以时间片作为我们拆分流数据的依据。一个时间片的数据对应一个 RDD 实例。
- **窗口长度（window length）**：一个窗口覆盖的流数据的时间长度。必须是批处理时间间隔的倍数，
- **滑动时间间隔**：前一个窗口到后一个窗口所经过的时间长度。必须是批处理时间间隔的倍数
- **Input DStream**：一个 input DStream 是一个特殊的 DStream，将 Spark Streaming 连接到一个外部数据源来读取数据。

13.1.2　Spark Streaming 处理流程

Spark Streaming 处理的数据流图如图 13-1 所示。



图 13-1　Spark Streaming 计算过程

Spark 的各个子框架，都是基于核心 Spark 的，Spark Streaming 在内部的处理机制是，接收实时流的数据，并根据一定的时间间隔拆分成一批批的数据，然后通过 Spark Engine 处理这些批数据（请参考图 13-2），最终得到处理后的一批批结果数据。

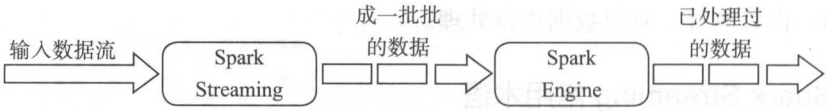


图 13-2 Spark Streaming 按批处理

DStream（Discretized Stream）作为 Spark Streaming 的基本抽象，它代表持续性的数据流。这些数据流既可以通过外部输入源来获取，也可以通过现有的 Dstream 的 transformation 操作来获得。在内部实现上，DStream 由一组时间序列上连续的 RDD 来表示。每个 RDD 都包含了自己特定时间间隔内的数据流。DStream 中在时间轴下生成离散的 RDD 序列如图 13-3 所示。

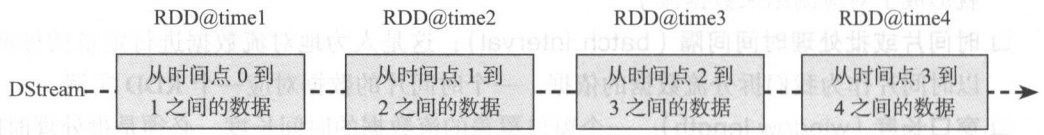


图 13-3 DStream 内部是 RDD 序列

对应的批数据，在 Spark 内核对应一个 RDD 实例，因此，对应流数据的 DStream 可以看成是一组 RDDs，即 RDD 的一个序列。通俗点理解的话，在流数据分成一批一批后，通过一个先进先出的队列，然后 Spark Engine 从该队列中依次取出一个个批数据，把批数据封装成一个 RDD，然后进行处理。

13.2 Dstream 操作

RDD 有很多操作和转换，与 RDD 类似，DStream 也提供了自己的一系列操作方法，本节主要介绍如何操作 DStream，包括输入、转换、修改状态及输出等。

13.2.1 Dstream 输入

在 Spark Streaming 中所有的操作都是基于流的，而输入源是一切操作的起点。

Spark Streaming 提供两种类型的流式输入数据源：

- 基础输入源：能直接应用于 StreamingContext API 输入源。例如，文件系统、Socket（套接字）连接和 Akka actors；
- 高级输入源：能应用于特定工具类的输入源，如 Kafka、Flume、Kinesis、Twitter 等，使用这些输入源需要导入一些额外依赖包。

13.2.2 Dstream 转换

DStream 转换操作是在一个或多个 DStream 上创建新的 DStream。DStream 支持多种转

换，详细请参考表 13-1 所示的内容。

表 13-1　Dstream 转换

转换	含义
map（func）	源 DStream 的每个元素通过函数 func 返回一个新的 DStream
flatMap（func）	类似与 map 操作，不同的是每个输入元素可以被映射出 0 或者更多的输出元素
filter（func）	在源 DSTREAM 上经过 func 函数，返回仅为 true 的元素，最后返回一个新的 DStream
repartition（numPartitions）	通过参数 numPartitions 值来修改 DStream 的平行程度
union（otherStream）	返回一个包含源 DStream 与其他 DStream 的元素的一个新 DStream
count()	统计源 Dstream 中 RDD 的元素个数
reduce（func）	使用函数 func（有两个输入参数，一个输出参数）将源 DStream 中每个 RDD 的元素进行聚合操作，返回一个单元素 RDD 的新 DStream
countByValue()	计算 DStream 中每个 RDD 内的元素出现的频次并返回新的 DStream[（K，Long）]，其中 K 是 RDD 中元素的类型，Long 是元素出现的频次
reduceByKey（func, [numTasks]）	当调用一个类型为（K，V）键值对的 Dstream 时，返回一个新的类型为（K，V）键值对的 DStream, 其中键 K 一般不变，每个键的值 V 都是使用聚合函数 func 汇总。注意：默认情况下，使用 Spark 的默认并行度提交任务，可以通过配置 numTasks 设置不同的并行任务数
join（otherStream, [numTasks]）	当调用类型分别为（K，V）和（K，W）键值对的 2 个 DStream 时，返回一个类型为（K，（V，W））键值对的一个新 DStream
cogroup（otherStream, [numTasks]）	当调用的两个分别含有（K，V）和（K，W）键值对的 DStream 时，返回一个（K，Seq[V]，Seq[W]）类型的新的 DStream
transform（func）	通过对源 DStream 的每个 RDD 应用一个 RDD-to-RDD 函数，返回一个新的 DStream，支持在新的 Dstream 上做任意 RDD 操作
updateStateByKey（func）	返回一个新状态的 DStream, 其中每个键的状态根据键的前一个状态和键的新值应用给定函数 func 后进行更新。这个方法可以被用来维持每个键的任何状态数据

13.2.3　Dstream 修改

Spark Streaming 除提供一些基本操作，还提供一些状态操作。状态操作对进行数据多批次操作，包括基于窗口（window）的操作和 updateStateByKey（func）操作等，表 13-2 所示为主要状态操作。

表 13-2　Dstream 修改

转换	含义
window（windowLength，slideInterval）	返回一个基于源 DStream 窗口化的新 DStream
countByWindow（windowLength，slideInterval）	返回基于滑动窗口的 DStream 中的元素量
reduceByWindow（func，windowLength，slideInterval）	基于滑动窗口对源 DStream 中的元素进行聚合操作，得到一个新的 DStream

（续）

转换	含义
reduceByKeyAndWindow（func，windowLength，slideInterval，[numTasks]）	基于滑动窗口对（K，V）键值对的 DStream 的值按 K 使用聚合，函数 func 进行聚合操作，得到一个新的 DStream
reduceByKeyAndWindow（func，invFunc，windowLength，slideInterval，[numTasks]）	一个更高效的 reduceByKkeyAndWindow，每个窗口的 Reduce 值是通过先前窗口的 Reduce 值增量计算获取的
countByValueAndWindow（windowLength，slideInterval，[numTasks]）	基于滑动窗口计算源 DStream 中每个 RDD 内每个元素出现的频次并返回一个新的 DStream[（K，Long）]，其中 K 是 RDD 中元素的类型，Long 是元素频率

13.2.4 Dstream 输出

Spark Streaming 允许 DStream 的数据输出到外部系统，如数据库、文件系统等。由于输出操作使转换操作后的数据可以通过外部系统使用，同时输出操作触发所有 DStream 的转换操作的实际执行（类似于 RDD 操作）。表 13-3 列出了目前主要的输出操作。

表 13-3 Dstream 输出

转换	描述
print()	在 Driver 中打印出 DStream 中数据的前 10 个元素
saveAsTextFiles（prefix，[suffix]）	将 DStream 中的内容以文本的形式保存，其中每批次处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix] 的方式命名
saveAsObjectFiles（prefix，[suffix]）	将 DStream 中的内容按对象序列化并且以 SequenceFile 的格式保存。其中每批次处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix] 的方式命名
saveAsHadoopFiles(prefix，[suffix])	将 DStream 中的内容以文本的形式保存为 Hadoop 文件，其中每批次处理间隔内产生的文件以 prefix-TIME_IN_MS[.suffix] 的方式命名
foreachRDD（func）	最基本的输出操作，将 func 函数应用于 DStream 中的 RDD 上，这个操作会输出数据到外部系统，比如保存 RDD 到文件或者网络数据库等

13.3 Spark Streaming 应用实例

本章前几节我们介绍了 Spark Streaming 的一些基本原理及主要操作，本节我们通过一个实例来进一步理解 Spark Streaming 的原理及操作，这个实例为在线统计 Linux 一个终端工具 NetCat（简称为 NC）手工发送文本，Spark 通过 tcp socket 读取文本、使用 Spark SQL 统计词频，然后用 print 输出统计结果。使用 scala 进行词频统计，大家可参考 Spark 官网：http://spark.apache.org/。

这里我们以 Spark 交互式编程方式进行，需要启动两个客户端或进程，一个客户端启动 nc，另一个启动 Spark shell。

先启动 nc，端口为 9999。

```
nc -lk 9999
```

然后，以本地方式启动 spark shell。

```
// 导入类或包
import org.apache.spark.SparkConf
import org.apache.spark.rdd.RDD
import org.apache.spark.sql.SparkSession
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.{Seconds, StreamingContext, Time}
import spark.implicits._


// 创建一个间隔时间为 3 秒的 context
val ssc = new StreamingContext(sc, Seconds(3))
// 创建一个 socket stream，基于 master:9999
val lines = ssc.socketTextStream("master",9999)
val words = lines.flatMap(_.split(" "))
// 为便于使用 SQL 进行统计，把 DStream 的 RDD 转换为 DataFrame。
// 把 RDD[String] 转换为 RDD[case class]，最后转换为 DataFrame
case class Record(word: String)

words.foreachRDD { (rdd:RDD[String], time:Time) =>
val wordsDataFrame = rdd.map(w => Record(w)).toDF()

// 创建一个临时视图
wordsDataFrame.createOrReplaceTempView("words")
// 使用 SQL 进行统计
val wordCountsDataFrame = spark.sql("select word, count(*) as total from words group by word")
println(s"========= $time =========")
wordCountsDataFrame.show()
    }
ssc.start()
ssc.awaitTermination()
```

在启动了 nc 的界面输入：

```
ok ok m p py py py
```

在 spark shell 界面，可以看到如下输出：

```
========= 1494714360000 ms =========
+----+-----+
|word|total|
+----+-----+
|   m|    1|
|  ok|    2|
|   p|    1|
|  py|    3|
+----+-----+
```

13.4 Spark Streaming 在线学习实例

前面我们简单介绍一个利用 nc 产生文本数据，Spark Streaming 实时统计词频的一个实例，通过这个例子，我们对 Streaming 有个大致了解，它的源数据可以是实时产生、实时变化的，基于这个数据流，Spark Streaming 能实时进行统计词频信息，并输出到界面。

除了统计词频，实际上 Spark Streaming 还可以做在线机器学习工作，目前 Spark Streaming 支持 Streaming Linear Regression，Streaming KMeans 等，这节我们模拟一个在线学习线性回归的算法，源数据为多个文件，首先在一个文件中训练模型，然后在新数据上进行调整模型，对新数据进行预测等。

```
// 导入需要的类
import org.apache.spark.mllib.linalg.Vectors
import org.apache.spark.mllib.feature.StandardScaler
import breeze.linalg.DenseVector
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.StreamingLinearRegressionWithSGD
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._


// 交互式编程
val ssc = new StreamingContext(sc, Seconds(10))
val stream = ssc.textFileStream("file:///home/hadoop/data/streaming/traindir")

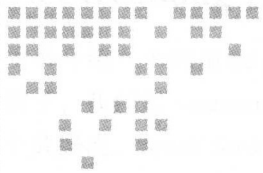
val NumFeatures = 11
val zeroVector = DenseVector.zeros[Double](NumFeatures)
val model = new StreamingLinearRegressionWithSGD()
.setInitialWeights(Vectors.dense(zeroVector.data))
.setNumIterations(20)
.setRegParam(0.8)
.setStepSize(0.01)


// 创建一个含标签的数据流
val labeledStream = stream.map { line =>
val split = line.split(";")
val y = split(11).toDouble
val features=split.slice(0,11).map(_.toDouble)
    LabeledPoint(label = y, features = Vectors.dense(features))
    }
// 在数据流上训练测试模型。
model.trainOn(labeledStream)
model.predictOnValues(labeledStream.map(lp => (lp.label, lp.features))) .print()
// 启动 Spark Streaming
```

```
ssc.start()
ssc.awaitTermination()
```

13.5　小结

前几章主要介绍了 Spark ML 对批量数据或离线数据的分析和处理，本章主要介绍 Spark Streamin 对在线数据或流式数据的处理及分析。首先对 Spark Streaming 的一些概念、输入源、Dstream 的一些转换、修改、输出作了简单介绍，然后，通过两个实例把这些内容结合在一起，进一步说明 Spark Streaming 在线统计、在线学习的具体使用。



Chapter 14 第 14 章

TensorFlowOnSpark 详解

前面我们介绍了 Spark MLlib 的多种机器学习算法，如分类、回归、聚类、推荐等，Spark 目前还缺乏对神经网络、深度学习的足够支持，但近几年市场对神经网络，尤其对深度学习热情高涨，成了当下很多企业的研究热点，缺失神经网络的支持，这或许也算是 Spark MLlib 尚欠不足之处吧。

不过好消息是 TensorFlow 这个深度学习框架，已经有了 Spark 接口，即 TensorFlowOnSpark。TensorFlow 是目前很热门的深度学习框架，是 Google 于 2015 年 11 月 9 日开源的第二代深度学习系统，也是 AlphaGo 的基础程序。

本章我们将介绍深度学习最好框架 TensorFlow 及 TensorFlowOnSpark，具体包括：

- TensorFlow 简介。
- TensorFlow 实现卷积神经网络。
- 分布式 TensorFlow。
- TensorFlowOnSpark 架构。
- TensorFlowOnSpark 实例。

14.1 TensorFlow 简介

TensorFlow 是谷歌基于 DistBelief 进行研发的第二代人工智能学习系统，采用数据流图（data flow graphs），用于数值计算的开源软件库。节点（nodes）在图中表示数学操作，图中的线（edges）则表示在节点间相互联系的多维数据数组，即 Tensor（张量），而 Flow（流）意味着基于数据流图的计算，TensorFlow 为张量从流图的一端流动到另一端计算过程。

TensorFlow 不只局限于神经网络，其数据流式图支持非常自由的算法表达，当然也可以轻松实现深度学习以外的机器学习算法。事实上，只要可以将计算表示成计算图的形式，就可以使用 TensorFlow。

TensorFlow 可被用于语音识别或图像识别等多项机器深度学习领域，TensorFlow 一大亮点是支持异构设备分布式计算，它能够在各个平台上自动运行模型，从手机、单个 CPU / GPU 到成百上千 GPU 卡组成的分布式系统。

14.1.1　TensorFlow 的安装

安装 TensorFlow，因本环境的 python2.7 采用 anaconda 来安装，故这里采用 conda 管理工具来安装 TensorFlow，目前 conda 默认安装版本为 TensorFlow 1.1。

```
conda  install  tensorflow
```

验证安装是否成功，可以通过导入 TensorFlow 来检验。

启动 ipython（或 python）：

```
import tensorflow as tf
```

14.1.2　TensorFlow 的发展

2015 年 11 月 9 日谷歌开源了人工智能系统 TensorFlow，并成为 2015 年最受关注的开源项目之一。TensorFlow 的开源大大降低了深度学习在各个行业中的应用难度。TensorFlow 的近期里程碑事件主要有：

2016 年 04 月：发布了分布式 TensorFlow 的 0.8 版本，把 DeepMind 模型迁移到 TensorFlow；

2016 年 06 月：TensorFlow v0.9 发布，改进了移动设备的支持；

2016 年 11 月：TensorFlow 开源一周年；

2017 年 2 月：TensorFlow v1.0 发布，增加了 Java、Go 的 API，以及专用的编译器和调试工具，同时 TensorFlow 1.0 引入了一个高级 API，包含 tf.layers、tf.metrics 和 tf.losses 模块。还宣布增加了一个新的 tf.keras 模块，它与另一个流行的高级神经网络库 Keras 完全兼容。

2017 年 4 月：TensorFlow v1.1 发布，为 Windows 添加 Java API 支持，添加 tf.spectral 模块，Keras 2 API 等；

2017 年 6 月：TensorFlow v1.2 发布，包括 API 的重要变化、contrib API 的变化和 Bug 修复及其他改变等。

14.1.3　TensorFlow 的特点

❑ 高度的灵活性：TensorFlow 采用数据流图，用于数值计算的开源软件库。只要计算

能表示为一个数据流图，你就可以使用 TensorFlow。

- 真正的可移植性：TensorFlow 在 CPU 和 GPU 上运行，可以运行在台式机、服务器、云服务器、手机移动设备、Docker 容器里等等。
- 将科研和产品联系在一起：过去如果要将科研中的机器学习想法用到产品中，需要大量的代码重写工作。TensorFlow 改变了这一点。使用 TensorFlow 可以让应用型研究者将想法迅速运用到产品中，也可以让学术性研究者更直接地彼此分享代码，产品团队则用 TensorFlow 来训练和使用计算模型，并直接提供给在线用户，从而提高科研产出率。
- 自动求微分：基于梯度的机器学习算法会受益于 TensorFlow 自动求微分的能力。使用 TensorFlow，只需要定义预测模型的结构，将这个结构和目标函数（objective function）结合在一起，并添加数据，TensorFlow 将自动为你计算相关的微分导数。
- 多语言支持：TensorFlow 有一个合理的 C++ 使用界面，也有一个易用的 Python 使用界面来构建和执行你的 graphs。你可以直接写 Python/C++ 程序，也可以用交互式的 IPython 界面来用 TensorFlow 尝试这些想法，也可以使用 Go、Java、Lua、Javascript 或者是 R 等语言。
- 性能最优化：如果你有一个 32 个 CPU 内核、4 个 GPU 显卡的工作站，想要将你工作站的计算潜能全发挥出来，由于 TensorFlow 给予了线程、队列、异步操作等以最佳的支持，TensorFlow 让你可以将你手边硬件的计算潜能全部发挥出来。你可以自由地将 TensorFlow 图中的计算元素分配到不同设备上，充分利用这些资源。

表 14-1 所示为 TensorFlow 的一些主要技术特征。

表 14-1 TensorFlow 的主要技术特征

编辑模型	Dataflow-like model（数据流模型）
语言	Python、C++、Go、Rust、Haskell、Java（还有非官方的 JavaScript、R 等）
部署	Code once、run everywhere（一次编写，各处运行）
计算资源	CPU(Linux、Mac、Windows、Android，iOS)GPU(Linux、Mac、Windows)TPU(Tensor Processing Unit)
实现方式	Local Implementation（单机实现）Distributed Implementation（分布式实现）
平台支持	Google Cloud Platform（谷歌云平台）Hadoop File System（Hadoop 分布式文件系统）
数学表达	Math Graph Expression（数学计算图表达）Auto Differentiation（自动微分）

14.1.4 TensorFlow 编程模型

TensorFlow 是一个采用数据流图（Data Flow Graph），用于数值计算的开源软件库。它用结点（node）和边（edge）的有向无环图（DAG）来描述数学计算。节点一般用来表示施加的数学操作（tf.Operation），但也可以表示数据输入（feed in）的起点 / 输出（push out）的终点，或者是读取 / 写入持久变量（persistent variable）的终点。边表示节点之间的输入 / 输出关系。这些数据“边”可以输运“size 可动态调整”的多维数据数组，即“张量”(tensor)。

张量从图中流过的直观图像是这个工具取名为“TensorFlow”的原因。

TensorFlow 如何工作？我们通过一个简单的实例进行说明，为计算 x+y，你需要创建如图 14-1 所示的数据流图。

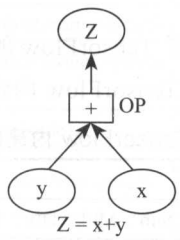


图 14-1　计算 x+y 的数据流图

数据流图（图 14-1）的详细步骤如下：

1）定义 x= [1,3,5]，y =[2,4,7]，并创建恒定的张量：

```
import tensorflow as tf
x = tf.constant([1,3,5])
y = tf.constant([2,4,7])
```

2）定义操作：

```
op = tf.add(x,y)
```

3）张量和操作都有了，接下来就是创建图：

```
my_graph = tf.Graph()
```


注意　这一步不是必须的，在创建回话时，系统将自动创建一个默认图。

4）为了运行这图你将需要创建一个回话（tf.Session），一个 tf.Session 对象封装了操作对象执行的环境，为了做到这一点，我们需要定义在会话中将要用到哪一张图：

```
with tf.Session(graph=my_graph) as sess:
    x = tf.constant([1,3,5])
    y = tf.constant([2,4,7])
    op = tf.add(x,y)
```

5）想要执行这个操作，要用到 tf.Session.run() 这个方法：

```
import tensorflow as tf
        my_graph = tf.Graph()
        with tf.Session(graph=my_graph) as sess:
        x = tf.constant([1,3,5])
        y = tf.constant([2,4,7])
        op = tf.add(x,y)
```

```
result = sess.run(fetches=op)
print(result)
```

6）运行结果：

[3 7 12]

上面我们通过简单的示例，说明了 TensorFlow 的运行方式，其中 add 为运算操作，类似的操作还有很多，如表 14-2 所示为 TensorFlow 内建的常用运算操作。

表 14-2 TensorFlow 内建的运算操作

类型	示例
数学运算	Add、Sub、Mul、Div、Exp、Log、Greater、Less、Equal
向量运算	Concat、Slice、Splot、Constant、Rank、Shape、Shuffle
矩阵运算	Matmul、MatrixInverse、MatrixDeterminant
带状态的运算	Variable、Assign、AssignAdd
神经网络组件	SoftMax、Sigmoid、ReLU、Convolution2D、MaxPooling
存储、恢复	Save、Restore
队列及同步运算	Enqueue、Dequeue、MutexAcquire、MutexRelease
控制张量流动	Merge、Switch、Enter、Leave、NextIteration

从上例我们可以看到 TensorFlow 有不少概念，这些概念或名称有些我们在其他系统中看到或使用过，但在 TensorFlow 架构中，很多概念的用途并不一样，表 14-3 总结了一些主要的概念，便于大家参考。

表 14-3 TensorFlow 常用概念

类型	描述	用途
Session	会话	图必须在称之为“会话”的上下文中执行。会话将图的 op 分发到诸如 CPU 或者 GPU 上计算
Graph	描述计算过程	必须在 Session 中启动
tensor	数据或张量	数据类型之一，代表多维数组
op	操作	图中的节点被称之为 op，一个 op 获得 0 或者多个 Tensor，执行计算，产生 0 或者多个 Tensor
Variable	变量	数据类型之一，运行过程中可以被改变，用于维护状态
feed	赋值	为 op 的 tensor 赋值
fetch	取值	从 op 的 tensor 中取值
Constant	常量	数据类型之一，不可变

14.1.5 TensorFlow 常用函数

TensorFlow 的函数有很多，类别各不相同，如数据类型转换、变量定义、激活函数、卷积函数等，这里我们选择一些常用或本章后续用到的一些函数，供大家参考，如表 14-4 所示。

表 14-4　TensorFlow 常用函数

类　别	函　数	含　义
tf.Tensor	tf.Tensor.eval(feed_dict=None, session=None)	在会话中求 tensor 的值。需要使用 with sess.as_default() 或者 eval(session=sess)
占位符	tf.placeholder(dtype, shape=None, name=None)	为一个 tensor 插入一个占位符
会话管理	tf.Session.run(fetches, feed_dict=None, options=None, run_metadata=None)	运行 fetches 中的操作节点并求其值
形状操作	tf.reshape(tensor, shape, name=None)	改变 tensor 的形状
矩阵相关运算	tf.transpose(a, perm=None, name= 'transpose ')	调换 tensor 的维度顺序，按照列表 perm 的维度排列调换 tensor 顺序
序列比较与索引提取	tf.argmin(input, dimension, name=None)	返回 input 最小值的索引 index
激活函数	tf.nn.relu(features, name=None)	整流函数：max(features, 0)
	tf.nn.dropout(x, keep_prob, noise_shape=None, seed=None, name=None)	计算 dropout，keep_prob 为 keep 概率，noise_shape 为噪声的 shape
	tf.sigmoid(x, name=None)	y = 1 / (1 + exp(-x))
	tf.tanh(x, name=None)	双曲线切线激活函数
卷积函数	tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=None, data_format=None, name=None)	结果返回一个 Tensor，这个输出，就是我们常说的 feature map，shape 仍然是 [batch, height, width, channels] 这种形式
池化函数	tf.nn.avg_pool(value, ksize, strides, padding, data_format= 'NHWC ', name=None)	平均方式池化
	tf.nn.max_pool(value, ksize, strides, padding, data_format= 'NHWC ', name=None)	最大值方法池化
数据标准化	tf.nn.l2_normalize(x, dim, epsilon=1e-12, name=None)	对维度 dim 进行 L2 范式标准化
分类函数	tf.nn.softmax(logits, name=None)	计算 softmax，softmax[i, j] = exp(logits[i, j]) / sum_j(exp(logits[i, j]))
	tf.nn.softmax_cross_entropy_with_logits(logits, labels, name=None)	计算 logits 和 labels 的 softmax 交叉熵，logits, labels 必须为相同的 shape 与数据类型
循环神经网络	tf.nn.rnn(cell, inputs, initial_state=None, dtype=None, sequence_length=None, scope=None)	基于 RNNCell 类的实例 cell 建立循环神经网络
优化器	class tf.train.GradientDescentOptimizer	使用梯度下降算法的 Optimizer
	class tf.train.AdamOptimizer	使用 Adam 算法的 Optimizer

14.1.6　TensorFlow 运行原理

TensorFlow 有一个重要组件 client，即客户端，此外，还有 master、worker，这些有点类似 Spark 的结构。它通过 Session 的接口与 master 及多个 worker 相连，其中每一个 worker 可以与多个硬件设备（device）相连，比如 CPU 或 GPU，并负责管理这些硬件。而 master 则负责管理所有 worker 按流程执行计算图。

TensorFlow 有单机模式和分布式模式两种实现，其中单机指 client、master、worker 全

部在一台机器上的同一个进程中；分布式的版本允许 client、master、worker 在不同机器的不同进程中，同时由集群调度系统统一管理各项任务。如图 14-2 所示为单机版和分布式版本的实现原理图。

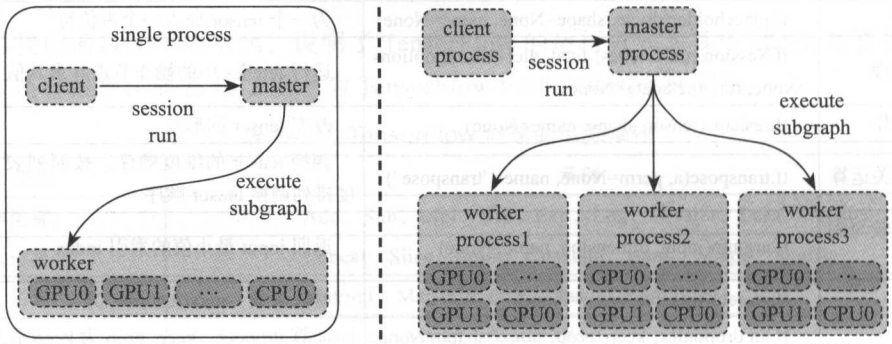


图 14-2 TensorFlow 单机版本和分布式运行原理

14.1.7 TensorFlow 系统架构

如图 14-3 所示为 TensorFlow 的系统架构，从底向上分为设备管理和通信层、数据操作层、图计算层、API 接口层、应用层。其中设备管理和通信层、数据操作层、图计算层是 TensorFlow 的核心层。

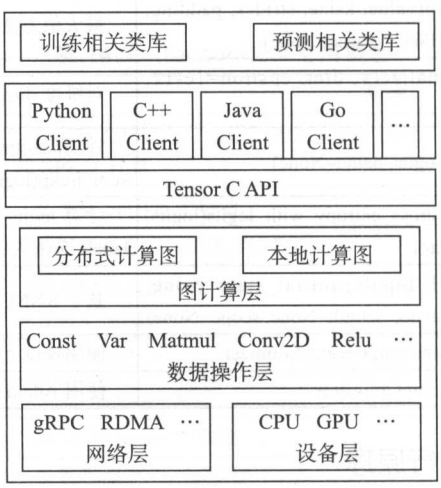


图 14-3 TensorFlow 系统架构

- 底层设备通信层负责网络通信和设备管理：设备管理可以实现 TensorFlow 设备异构的特性，支持 CPU、GPU、Mobile 等不同设备。网络通信依赖 gRPC 通信协议实现不同设备间的数据传输和更新。

- 第二层为数据操作层实现：这些操作以 Tensor 为处理对象，依赖网络通信和设备内存分配，实现了各种 Tensor 操作或计算。操作不仅包含 MatMul 等计算操作，还包含 Queue 等非计算操作。
- 第三层是图计算层（Graph）：包含本地计算流图和分布式计算流图的实现，Graph 模块包含 Graph 的创建、编译、优化和执行等部分，Graph 中每个节点都是操作类型表示。
- 第四层是 API 接口层：Tensor C API 是对 TensorFlow 功能模块的接口封装，便于其他语言平台调用。
- 第四层以上是应用层：不同编程语言在应用层通过 API 接口层调用 TensorFlow 核心功能实现相关应用。

14.2　TensorFlow 实现卷积神经网络

神经网络可为机器学习中最活跃的领域之一，尤其代表深度学习的卷积神经（Convolutional Neural Network,CNN）、循环神经网络（Recurrent Neural Network，RNN）更是炙手可热。

14.2.1　卷积神经网络简介

卷积神经网络是人工神经网络的一种，已成为图像识别、视频处理、语音分析等领域的研究热点。它的权值共享网络结构使之更类似于生物神经网络，减少了权值的数量，降低了网络模型的复杂度，避免因参数太多导致过拟合。

卷积神经网络在训练时自动提取最有效特征，使图像可以直接作为输入，避免了传统机器学习算法中复杂的特征提取和数据重建过程。

在卷积神经网络中，第一个卷积层直接接受图像像素级的输入，每一个卷积操作只处理一小块图像，进行卷积变化后再传到后面的网络层。每一层卷积都会提取最有效的特征。这种方法可以提取图像中最基础的特征（图像都是一些基础特征构成的，音频也有其基础特征），例如不同方向的边或者角点，而后进行组合和抽象等更高级的特征。因此，CNN 可以应对各种情况，对平移、比例缩放、倾斜或者共他形式的变形具有高度不变性。

有几个概念，如共享权重，自动提取特征等对初学者可能不易理解，这里我们通过一个示例进行说明。

假设有一个原 5 × 5 像素的一个图像（当然一般图像的像素远大于这个数，此外还要考虑颜色通道等），现在对该图像进行卷积处理，假设卷积核大小为 3 × 3（一个卷积层中可以有多个卷积核），移动步长（strides）为 1，那么卷积核（图 14-4 中的深色小窗口）在 5 × 5 图片，按步长从左到右，从上到下移动时，卷积核在二维平面上平移，并且卷积核的每个

元素（权重）与被卷积图像对应位置相乘，再求和。通过卷积核的不断移动，就可以生成一个新的图像，如图 14-4 右边 3 × 3 的图像。

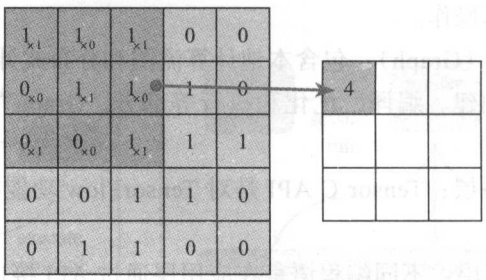


图 14-4　提取第一个特征值 4 的示意图

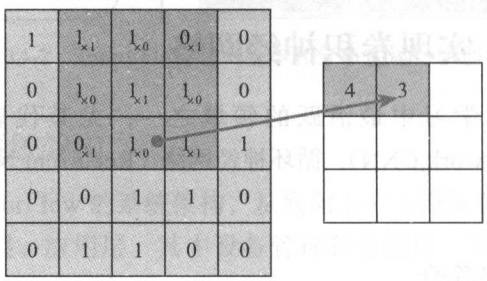


图 14-5　提取第二个特征值 3 的示意图

黄色小窗口在移动过程中，因它们同属一个卷积核，其权重不会随着移动而改变（即所谓的权重共享），都是：

$$\begin{matrix}1 & 0 & 1\\ 0 & 1 & 0\\ 1 & 0 & 1\end{matrix}$$

图 14-5 右边窗口中的 4 和 3，分别是深色窗口中的值乘以对应权重得到的：

$4=1\times1+1\times0+1\times1+0\times0+1\times1+1\times0+0\times1+0\times0+1\times1$

$3=1\times1+1\times0+0\times1+1\times0+1\times1+1\times0+0\times1+1\times0+1\times1$

深色窗口按步长每移动一次就可得到右边对应的一个值，共有 9 中移动方法，所以我们能得到 9 个特征值。

14.2.2　卷积神经网络的发展历程

深度学习（Deep Learning）尤其是卷积神经网络（CNN）已成为近几年来模式识别的研究重点，受到人们越来越多的关注，并产生了几种变种。不管如何变化，但是万变不离其宗，那些在深度学习发展过程中起关键作用的经典文献，这里依据时间线索，对 CNN 发展过程中出现的一些经典文献做一些梳理，方便大家在研究 CNN 时追本溯源，了解 CNN 的发展历史，了解大师们为推进 CNN 所做出的杰出贡献。如图 14-6 所示为 CNN 在发展过程

中的一些具有里程碑意义的事件和文献。

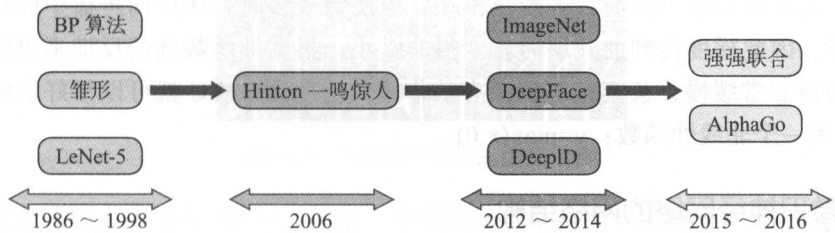


图 14-6　CNN 发展重要里程碑

对于 CNN 最早可以追溯到 1986 年 BP 算法的提出，1989 年 LeCun 将其应用到多层神经网络中，直到 1998 年 LeCun 提出 LeNet-5 模型，神经网络的雏形基本形成。在接下来近十年的时间里，卷积神经网络的相关研究处于低谷，原因有两个：一是研究人员意识到多层神经网络在进行 BP 训练时的计算量极其之大，以当时的硬件计算能力完全不可能实现；二是包括 SVM 在内的浅层机器学习算法也渐渐开始崭露头角。

2006 年，Hinton 终于一鸣惊人，在《科学》上发表文章，CNN 再度觉醒，并取得长足发展。2012 年，ImageNet 大赛上 CNN 夺冠，2014 年，谷歌研发出 20 层的 VGG 模型。同年，DeepFace、DeepID 模型横空出世，直接将 LFW 数据库上的人脸识别、人脸认证的正确率刷到 99.75%，几乎超越人类。2015 年深度学习领域的三巨头 LeCun、Bengio、Hinton 联手在 Nature 上发表综述对 DeepLearning 进行科普。2016 年 3 月 AlphaGo 打败李世石，使深度学习再一次点燃人们心中的热情。如图 14-7 所示为 CNN 的以技术为线索的发展历程。

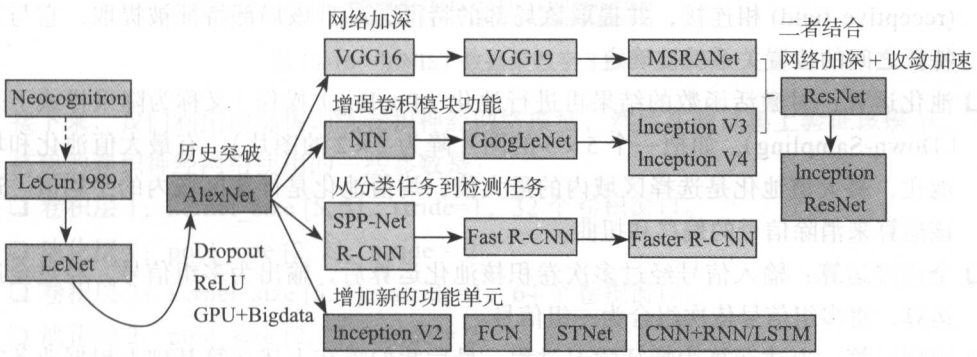


图 14-7　CNN 技术发展历程

其中有两个关键技术的突破——dropout 和 ReLU。dropout 是针对深度学习中过拟合问题，由 Hinton 教授团队提出的一个解决方案，基本思想是在训练时，将神经网络某一层的输出节点数据随机丢弃一部分。通过这种方式，将创建很多新样本，同时减少特征数量，以此避免过拟合的发生。

而 ReLU 是针对深度学习中梯度弥散问题，由 Hinton 教授团队提出的一个解决方案。

神经网络训练一般都使用 Sigmoid 作为激活函数，这种激活函数对于一般神经网络来说效果比较好，尤其在层数不多时，表现还不错，但层数较多时，在反向传播中的梯度将呈现指数级下降，因此梯度传到前几层时就变得非常小，根据训练数据的反馈来更新神经网络的参数就变得非常缓慢，甚至起不到应有的作用。但使用 ReLU 就可以很好地解决这类问题，ReLU 为一个非线性函数：y=max(x,0)。

14.2.3 卷积神经网络的网络结构

卷积神经网络经过多年发展，出现了多种技术，但基本网络结构相似，一般包括：卷积运算、池化运算、全连接运算和识别运算，如图 14-8 所示。

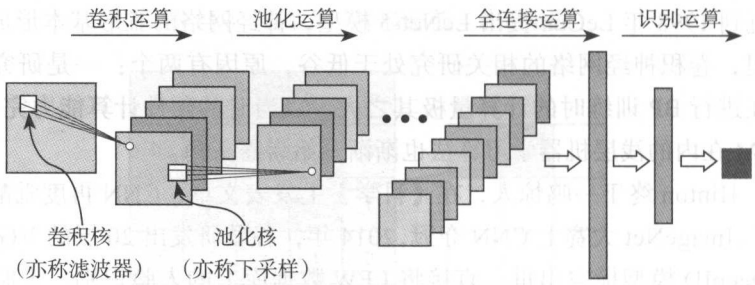


图 14-8 卷积神经网络结构

- 卷积运算：前一层的特征图与卷积核进行卷积运算，运算的结果再加偏移置，经过激活函数后形成这一层的神经元，每个神经元的输入与前一层的局部感受野 (receptive field) 相连接，并提取该局部的特征，一旦该局部特征被提取，它与其他特征之间的位置关系就被确定。
- 池化运算：对激活函数的结果再进行池化（Pooling）操作（又称为降采样或下采样（Down-Sampling），如把一个 5×5 的图片降为 2×2 的图片）。有最大值池化和均值池化，最大值池化是选择区域内的最大值，均值池化是计算区域内的平均值。通过该运算来消除信号的偏移和扭曲。
- 全连接运算：输入信号经过多次卷积核池化运算后，输出为多组信号，经过全连接运算，将多组信号依次组合为一组信号。
- 识别运算：上述运算为特征学习过程，最后我们需在上述运算基础上根据业务需求（分类或回归问题）增加一层网络用于分类或回归计算。

14.2.4 TensorFlow 实现卷积神经网络

本节通过一个实例来说明如何通过 TensorFlow 创建一个卷积神经网络架构，使用 MNIST 数据集，这一个非常有名的手写数字识别数据集，有 60000 张图片作为训练集，10000 张图片为测试集，每张图为一个手写的 0~9 的数字。如图 14-9 所示。

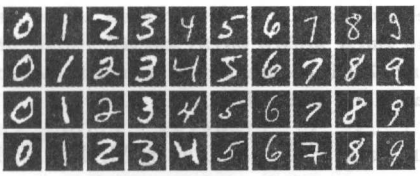


图 14-9　MNIST 数据集图片示例

其中每张图的大小均为 28*28，这里大小指的是像素。例如，数字 3 所对应的像素矩阵如图 14-10 所示。

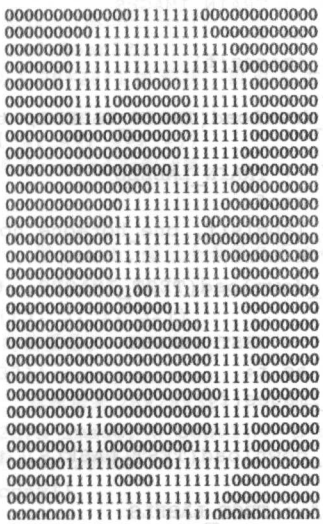


图 14-10　MNIST 数据集数字 3 图片及像素

接下来，我们利用训练集训练卷积神经网络模型，然后在测试集上验证该模型。

搭建的卷积神经网络使用的一些参数是：

- 卷积层 1：kernel_size [5, 5]，stride=1，32 个卷积窗口。
- 池化层 1：pool_size [2, 2]，stride = 2。
- 卷积层 2：kernel_size [5, 5]，stride=1，64 个卷积窗口。
- 池化层 2：pool_size [2, 2]，stride = 2。
- 全连接层：1024 个特征，使用 dropout 减少过拟合。
- 输出层：使用 softmax 进行分类。

1. 导入数据

首先启动 Ipython，进入交互计算环境，当然直接启动 Python 也可，然后通过 TensorFlow 自带的函数读取图片数据。

```
import tensorflow as tf
```

```
    from tensorflow.examples.tutorials.mnist import input_data
    mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
    ~/anaconda2/lib/python2.7/site-packages/tensorflow/contrib/learn/python/learn/
datasets/mnist.py 中函数 read_data_sets 四个 local_file
```

如果无法直接通过 input_data 下载，可以先把 MNIST 数据下载，然后，修改 python/learn/datasets/mnist.py 文件中 read_data_sets 函数中 4 个 local_file 的值。

具体如下，注释原来的 local_file，新增 4 行 local_file。

```
    #local_file = base.maybe_download(TRAIN_IMAGES, train_dir,SOURCE_URL + TRAIN_
IMAGES)
    local_file = train_dir + "/" + TRAIN_IMAGES
    with open(local_file, 'rb') as f:
        train_images = extract_images(f)

    #local_file = base.maybe_download(TRAIN_LABELS, train_dir,SOURCE_URL + TRAIN_
LABELS)
    local_file = train_dir + "/" + TRAIN_LABELS
    with open(local_file, 'rb') as f:
        train_labels = extract_labels(f, one_hot=one_hot)

    #local_file = base.maybe_download(TEST_IMAGES, train_dir,SOURCE_URL + TEST_
IMAGES)
    local_file = train_dir + "/" + TEST_IMAGES
    with open(local_file, 'rb') as f:
        test_images = extract_images(f)

    #local_file = base.maybe_download(TEST_LABELS, train_dir,SOURCE_URL + TEST_
LABELS)
    local_file = train_dir + "/" + TEST_LABELS
```

更加数据实际存放路径，修改 read_data_sets 中读取文件路径。

```
    mnist = input_data.read_data_sets("./TensorFlowOnSpark/mnist", one_hot=True)
    # 创建交互式 session
    sess = tf.InteractiveSession()
```

2. 权重初始化

```
    # 正态分布，标准差为 0.1，默认最大为 1，最小为 -1，均值为 0
    def weight_variable(shape):
            initial = tf.truncated_normal(shape, stddev=0.1)
            return tf.Variable(initial)
    # 创建一个结构为 shape 矩阵也可以说是数组 shape 声明其行列，初始化所有值为 0.1
    def bias_variable(shape):
            initial = tf.constant(0.1, shape=shape)
            return tf.Variable(initial)
```

3. 构建卷积神经网络结构

```
    # 卷积遍历各方向步数为 1,SAME: 边缘外自动补 0，遍历相乘
    def conv2d(x, W):
```

```
        return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
    # 池化卷积结果(conv2d)池化层采用 kernel 大小为 2*2,步数也为 2,周围补 0,取最大值。数据量缩小了 4 倍
    def max_pool_2x2(x):
        return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1], padding='SAME')

    # 定义输入输出结构

    # 声明一个占位符,None 表示输入图片的数量不定,28*28 图片分辨率
    xs = tf.placeholder(tf.float32, [None, 28*28])
    # 类别是 0-9 总共 10 个类别,对应输出分类结果
    ys = tf.placeholder(tf.float32, [None, 10])
    keep_prob = tf.placeholder(tf.float32)
    # x_image 又把 xs reshape 成了 28*28*1 的形状,因为是灰色图片,所以通道是 1.作为训练时的 input,-1 代表图片数量不定
    x_image = tf.reshape(xs, [-1, 28, 28, 1])
    # 搭建网络,定义算法公式,也就是 forward 时的计算

    ## 第一层卷积操作 ##
    # 第一二参数值得卷积核尺寸大小,即 patch,第三个参数是图像通道数,第四个参数是卷积核的数目,代表会出现多少个卷积特征图像;
    W_conv1 = weight_variable([5, 5, 1, 32])
    # 对于每一个卷积核都有一个对应的偏置量。
    b_conv1 = bias_variable([32])
    # 图片乘以卷积核,并加上偏执量,卷积结果 28x28x32
    h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
    # 池化结果 14x14x32 卷积结果乘以池化卷积核
    h_pool1 = max_pool_2x2(h_conv1)

    ## 第二层卷积操作 ##
    # 32 通道卷积,卷积出 64 个特征
    w_conv2 = weight_variable([5,5,32,64])
    # 64 个偏执数据
    b_conv2  = bias_variable([64])
    # 注意 h_pool1 是上一层的池化结果,# 卷积结果 14x14x64
    h_conv2 = tf.nn.relu(conv2d(h_pool1,w_conv2)+b_conv2)
    # 池化结果 7x7x64
    h_pool2 = max_pool_2x2(h_conv2)
    # 原图像尺寸 28*28,第一轮图像缩小为 14*14,共有 32 张,第二轮后图像缩小为 7*7,共有 64 张

    ## 第三层全连接操作
    # 二维张量,第一个参数 7*7*64 的 patch,第二个参数代表卷积个数共 1024 个
    W_fc1 = weight_variable([7*7*64, 1024])
    # 1024 个偏执数据
    b_fc1 = bias_variable([1024])
    # 将第二层卷积池化结果 reshape 成只有一行 7*7*64 个数据 # [n_samples, 7, 7, 64] ->> [n_samples, 7*7*64]
    h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
    # 卷积操作,结果是 1*1*1024,matmul 实现最基本的矩阵相乘。
```

```
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# dropout 操作，减少过拟合。对卷积结果执行 dropout 操作。
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1,keep_prob)
## 第四层输出操作 ##
# 二维张量，1*1024 矩阵卷积，共 10 个卷积，对应我们开始的 ys 长度为 10
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
# 最后的分类，结果为 1*1*10 softmax
y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
#定义交叉熵为 loss 函数，采用 Adam 方法优化 loss。
cross_entropy = -tf.reduce_sum(ys * tf.log(y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

4. 训练评估模型

```
#模型训练及评测
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(ys,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
tf.global_variables_initializer().run()

for i in range(2000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={xs:batch[0], ys: batch[1],
keep_prob: 1.0})
        print("step %d, training accuracy %g"%(i, train_accuracy))
    train_step.run(feed_dict={xs: batch[0], ys: batch[1], keep_prob: 0.5})
```

这里只迭代了 2000 次，运行结果如下面代码所示（如果迭代 20000 次，在测试集上的精度可达 99.2%）。

```
## -- End pasted text --
step 0, training accuracy 0.14
step 100, training accuracy 0.86
step 200, training accuracy 0.94
step 300, training accuracy 0.94
step 400, training accuracy 0.94
step 500, training accuracy 0.98
step 600, training accuracy 0.94
step 700, training accuracy 0.94
step 800, training accuracy 0.98
step 900, training accuracy 0.98
step 1000, training accuracy 1
step 1100, training accuracy 0.94
step 1200, training accuracy 0.98
step 1300, training accuracy 0.96
step 1400, training accuracy 0.92
step 1500, training accuracy 0.96
step 1600, training accuracy 0.96
step 1700, training accuracy 1
```

```
    step 1800, training accuracy 1
    step 1900, training accuracy 0.96
```

在测试集上，测试模型精度：

```
    print("test accuracy %g"%accuracy.eval(feed_dict={xs: mnist.test.images, ys:
mnist.test.labels, keep_prob: 1.0}))
    test accuracy 0.9778
```

14.3　TensorFlow 实现循环神经网络

前面我们介绍了卷积神经网络（CNN），其核心在于构建特征过滤器，以处理输入图片中像素的空间位置信息。在自然语言处理中，很多与上下文有关或与时间序列有关，这类问题使用卷积神经网络不太适合，不过我们可以通过循环神经网络来处理，循环神经网络特别适合于含有时间序列或上下文联系的模型。这节我们将介绍循环神经网络（Recurrent Neural Network，RNN），循环神经网络在语音识别、语音翻译、机器翻译等领域成效显著，由于 RNN 涉及时间维度，在进行逐层反向传播时，易导致梯度消失或爆炸的问题，在 1997 年，Juergen Schmidhuber 和他的同事提出长短记忆（Long Short-Term Memory），从而解决了该问题，目前循环神经网络在很多领域取得了很大成功和突破。我们这节主要介绍 LSTM 神经网络。

14.3.1　循环神经网络简介

在传统的神经网络模型中，从输入层到隐含层再到输出层，层与层之间是全连接的，每层之间的节点是无连接的。但是这种普通的神经网络对于很多问题却无能为力。例如，你要预测句子的下一个单词是什么，一般需要用到前面的单词，因为一个句子中前后单词并不是独立的，即一个序列当前的输出与前面的输出也有关。具体的表现形式为网络会对前面的信息进行记忆并应用于当前输出的计算中，即隐藏层之间的节点不再是无连接而是有连接的，并且隐藏层的输入不仅包括输入层的输出，还包括上一时刻隐藏层的输出。

理论上，RNNs 能够对任何长度的序列数据进行处理。但是在实践中，为了降低复杂性，往往假设当前的状态只与前面的几个状态相关，如图 14-11 所示便是一个典型的 RNNs。

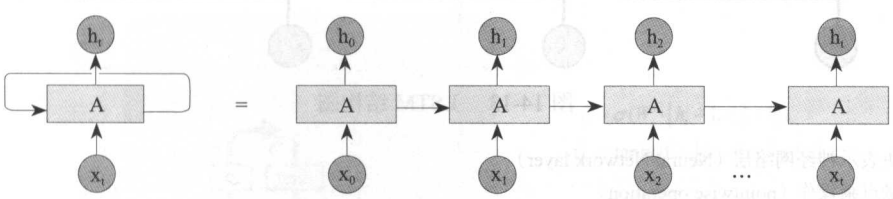


图 14-11　循环神经网络展开示意图

普通循环网络，虽然可以处理整个时间序列信息，但影响最大的还是最近或最后输入

的一些信息，更早的一些信号越来越弱，这个不足导致 RNN 在早期作用不明显。于 1997 年提出的 Long Short Term Mermory network（LSTM），很好地解决了这个问题，下节我们将对其进行介绍。

14.3.2 LSTM 循环神经网络简介

LSTM 是一种特殊的 RNNs，可以很好地解决长时依赖问题。那么它与常规神经网络有什么异同？首先我们来看 RNNs 的具体结构，如图 14-12 所示。

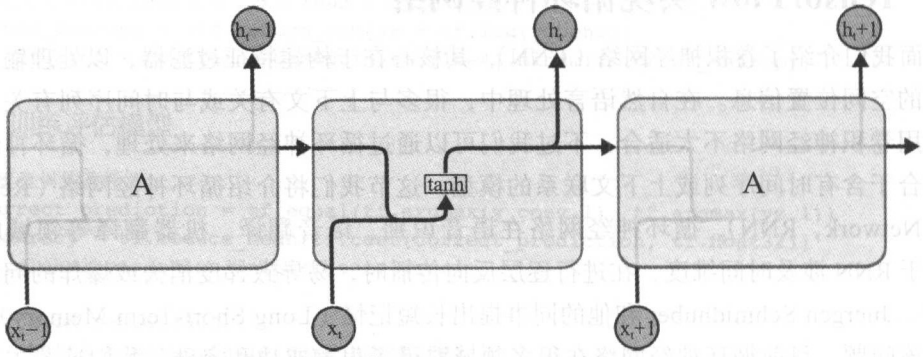


图 14-12 一般 RNN 结构图

所有的递归神经网络都是由重复神经网络模块构成的一条链，可以看到它的处理层非常简单，通常是一个单 tanh 层，通过当前输入及上一时刻的输出来得到当前输出。与神经网络相比，经过简单的改造，它已经可以利用上一时刻学习到的信息进行当前时刻的学习。

LSTM 的结构与上面相似，不同的是它的重复模块会比较复杂一点，它有四层结构，如图 14-13 所示。

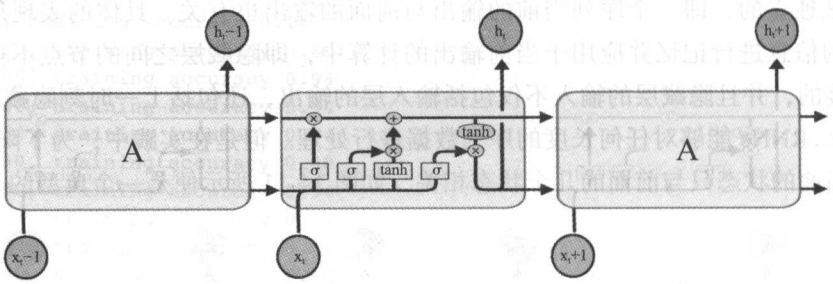


图 14-13 LSTM 结构图

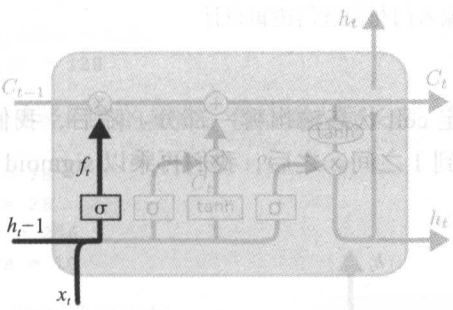
色块表示神经网络层（Neural Network layer）
向量点乘操作（pointwise operation）
向量（vector）从一个节点到另一个节点
合并操作（concatenate）
复制（copy）

LSTM 的关键就是矩形方框，被称为记忆块（memory block），主要包含了三个门（forget gate、input gate、output gate）与一个记忆单元（cell）。方框内上方的那条水平线，被称为单元状态（cell state），它就像一个传送带，可以控制信息传递给下一时刻。当 state 在这条传送带传递时，LSTM 单元可以对其添加或删除或修改信息，这些对信息的修改通过这些 Gate 来控制。这些 Gates 中包含一个 Sigmoid 层和一个向量点乘操作，这个 Sigmoid 的输出在 0 和 1 之间，它直接控制信息的传递比例。0 表示不允许，1 表示全部通过。每个 LSTM 单元都有这样的 3 个 Gates，通过这种方式 LSTM 就可实现长程记忆。

14.3.3　LSTM 循环神经网络分步说明

1）第一步：确定以往什么信息可以通过 cell state。

这个决定由忘记门（forget gate）通过 sigmoid 来控制，它会根据上一时刻的输出 h_{t-1} 和当前输入 x_t 来产生一个 [0,1] 区间的 f_t 值，来决定是否让上一时刻学到的信息 c_{t-1} 通过或部分通过。如图 14-14 所示。

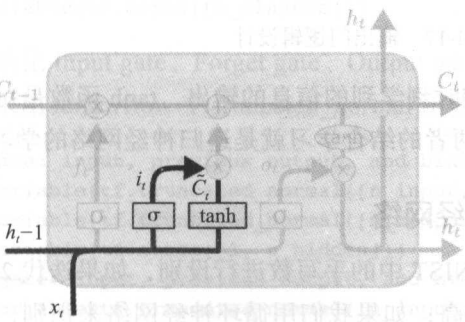


$$f_t = \sigma(W_f \cdot [h_t - 1, x_t] + b_f)$$

图 14-14　忘记门逻辑设计

2）第二步：产生新信息。

这一步包含两部分，第一个是输入门（input gate）通过 sigmoid 来决定哪些值用来更新，第二个是一个 tanh 层用来生成新的候选值 c_t，它作为当前层产生的候选值可能会添加到 cell state 中。我们会把这两部分产生的值结合来进行更新，如图 14-15 所示。



$$i_t = \sigma(W_i \cdot [h_t - 1, x_t] + b_i)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_t - 1, x_t] + b_C)$$

图 14-15　输入门与候选门逻辑设计

3）第三步：更新老的 cell state。

首先，我们将老的 cell state（即 C_t-_1）乘以 f_t 来忘掉我们不需要的信息，然后再与 $i_t*\tilde{C}_t$ 相加，得到了候选值。把这两步结合起来就是丢掉不需要的信息，添加新信息的过程，如图 14-16 所示。

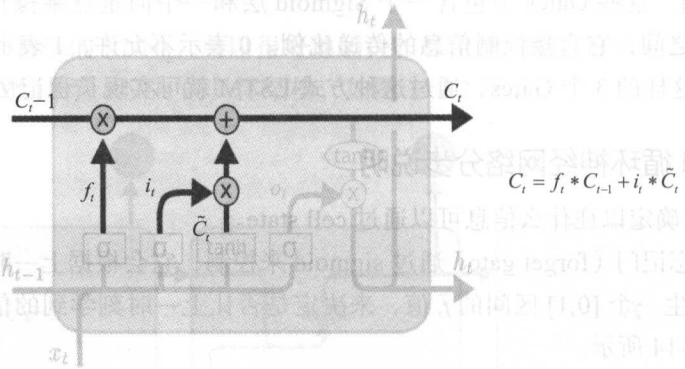


图 14-16 输入门与候选门逻辑设计

4）最后一步：决定模型的输出。

首先，通过运行一个 sigmoid 层决定 cell 状态输出哪一部分。随后，我们把 cell 状态通过 tanh 函数，将 C_t 输出值保持在 −1 到 1 之间。之后，我们再乘以 sigmoid 门的输出值，就可以得到结果了，如图 14-17 所示。

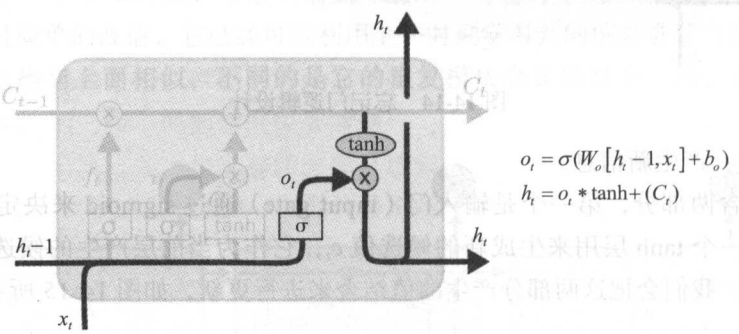


图 14-17 输出门逻辑设计

sigmoid 函数的输出是不考虑先前时刻学到的信息的输出，tanh 函数是对先前学到信息的压缩处理，起到稳定数值的作用，两者的结合学习就是递归神经网络的学习思想。

14.3.4 TensorFlow 实现循环神经网络

前面我们用卷积神经网络，对 MNIST 中的手写数进行设别，如果迭代 20000 次，精度可达到 99.2 左右，这个精度应该比较高；如果我们用循环神经网络来识别，是否可行？如

果可以，效果如何？

为了适合使用 RNN 来识别，每张图片大小为 28 × 28 像素，我们把每张图片的每一行（元素个数为 28）作为输入数据 n_inputs，把每一行（一张图片共 28 行）看成是与时间序列有关的步数 n_steps，这样图片的所有信息都用上了，而且适合使用 RNN 的应用场景。

启动 IPython，进入 IPython 的交互式界面，导入需要的库，并启动交互式会话。

```
import tensorflow as tf
import numpy as np
sess = tf.InteractiveSession()
```

加载数据，具体实现细节可参考 14.2.4 节，这里就不详细说明了。

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("./TensorFlowOnSpark/mnist", one_hot=True)
```

1. 构建模型

设置训练模型的超参数，学习速率，批量大小等。

```
learning_rate = 0.01
batch_size = 128
```

设置循环神经网络的参数，包括输入数长度，输入的步数，隐藏节点数，类别数等。

```
n_input = 28
n_steps = 28
n_hidden = 256
n_classes = 10
```

定义输入数据及权重等。

```
x = tf.placeholder(tf.float32, [None, n_steps, n_input])
y = tf.placeholder(tf.float32, [None, n_classes])
```

定义权重及初始化偏移量。

```
# Classifier weights and biases
w = tf.Variable(tf.truncated_normal([n_hidden, n_classes]))
b = tf.Variable(tf.zeros([n_classes]))
```

定义并初始化 Input gate、Forget gate、Output gate、Memory cell 等的输入数据、权重、偏移量，这里采用 tensorflow 中 truncated_normal 函数初始化相关参数值。

```
# Input gate: input, previous output, and bias
ix = tf.Variable(tf.truncated_normal([n_input, n_hidden], -0.1, 0.1))
im = tf.Variable(tf.truncated_normal([n_hidden, n_hidden], -0.1, 0.1))
ib = tf.Variable(tf.zeros([1, n_hidden]))
# Forget gate: input, previous output, and bias
fx = tf.Variable(tf.truncated_normal([n_input, n_hidden], -0.1, 0.1))
fm = tf.Variable(tf.truncated_normal([n_hidden, n_hidden], -0.1, 0.1))
```

```
fb = tf.Variable(tf.zeros([1, n_hidden]))
# Memory cell: input, state, and bias
cx = tf.Variable(tf.truncated_normal([n_input, n_hidden], -0.1, 0.1))
cm = tf.Variable(tf.truncated_normal([n_hidden, n_hidden], -0.1, 0.1))
cb = tf.Variable(tf.zeros([1, n_hidden]))
# Output gate: input, previous output, and bias
ox = tf.Variable(tf.truncated_normal([n_input, n_hidden], -0.1, 0.1))
om = tf.Variable(tf.truncated_normal([n_hidden, n_hidden], -0.1, 0.1))
ob = tf.Variable(tf.zeros([1, n_hidden]))
```

创建循环神经网络结构。

```
def LSTMRNN(x, n_steps, n_input, n_hidden, n_classes):
    # 定义 LSTM 单元
    def lstm_cell(i, o, state):
        input_gate = tf.sigmoid(tf.matmul(i, ix) + tf.matmul(o, im) + ib)
        forget_gate = tf.sigmoid(tf.matmul(i, fx) + tf.matmul(o, fm) + fb)
        update = tf.tanh(tf.matmul(i, cx) + tf.matmul(o, cm) + cb)
        state = forget_gate * state + input_gate * update
        output_gate = tf.sigmoid(tf.matmul(i, ox) +  tf.matmul(o, om) + ob)
        return output_gate * tf.tanh(state), state

    # 把状态线上的多个值串联起来
    outputs = list()
    state = tf.Variable(tf.zeros([batch_size, n_hidden]))
    output = tf.Variable(tf.zeros([batch_size, n_hidden]))

    # 输入数据 x 用函数 transpose 把第一个维度与第二个维度互换，使用 reshape 把 x
    # 变为 (n_steps*batch_size,n_input) 的形状，然后利用 split 把 x 拆成长度为 n_steps
    # 的列表，这样适合 LMTM 的输入格式。
    x = tf.transpose(x, [1, 0, 2])
    x = tf.reshape(x, [-1, n_input])
    x = tf.split(x, n_steps, 0)
    for i in x:
        output, state = lstm_cell(i, output, state)
        outputs.append(output)
    logits =tf.matmul(outputs[-1], w) + b
    return logits
```

2. 定义损失函数及优化器

```
pred = LSTMRNN(x, n_steps, n_input, n_hidden, n_classes)
cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred,
labels=y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate).minimize(cost)

correct_pred = tf.equal(tf.argmax(pred,1), tf.argmax(y,1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initializing the variables
```

```
init = tf.global_variables_initializer()

# Launch the graph
sess.run(init)
```

3. 训练数据及评估模型

```
for step in range(10000):
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    batch_x = batch_x.reshape((batch_size, n_steps, n_input))
    sess.run(optimizer, feed_dict={x: batch_x, y: batch_y})

    if step % 100 == 0:
        acc = sess.run(accuracy, feed_dict={x: batch_x, y: batch_y})
        loss = sess.run(cost, feed_dict={x: batch_x, y: batch_y})
        print "Iter " + str(step) + ", Minibatch Loss= " + "{:.6f}".format(loss)
+ ", Training Accuracy= " + "{:.5f}".format(acc)
print "Optimization Finished!"
```

运行结果，以下是最后批次的运行结果。

```
Iter 8000, Minibatch Loss= 0.085752, Training Accuracy= 0.97656
Iter 8100, Minibatch Loss= 0.065435, Training Accuracy= 0.96875
Iter 8200, Minibatch Loss= 0.088926, Training Accuracy= 0.97656
Iter 8300, Minibatch Loss= 0.039572, Training Accuracy= 1.00000
Iter 8400, Minibatch Loss= 0.050593, Training Accuracy= 0.98438
Iter 8500, Minibatch Loss= 0.030424, Training Accuracy= 0.99219
Iter 8600, Minibatch Loss= 0.026174, Training Accuracy= 0.99219
Iter 8700, Minibatch Loss= 0.045043, Training Accuracy= 0.98438
Iter 8800, Minibatch Loss= 0.031143, Training Accuracy= 0.98438
Iter 8900, Minibatch Loss= 0.055115, Training Accuracy= 0.99219
Iter 9000, Minibatch Loss= 0.061676, Training Accuracy= 0.98438
Iter 9100, Minibatch Loss= 0.123581, Training Accuracy= 0.97656
Iter 9200, Minibatch Loss= 0.057620, Training Accuracy= 0.98438
Iter 9300, Minibatch Loss= 0.043013, Training Accuracy= 0.99219
Iter 9400, Minibatch Loss= 0.067405, Training Accuracy= 0.98438
Iter 9500, Minibatch Loss= 0.020679, Training Accuracy= 1.00000
Iter 9600, Minibatch Loss= 0.079038, Training Accuracy= 0.98438
Iter 9700, Minibatch Loss= 0.080076, Training Accuracy= 0.97656
Iter 9800, Minibatch Loss= 0.010582, Training Accuracy= 1.00000
Iter 9900, Minibatch Loss= 0.019426, Training Accuracy= 1.00000
Optimization Finished!
```

在测试集上验证模型：

```
# Calculate accuracy for 128 mnist test images
test_len = batch_size
test_data = mnist.test.images[:test_len].reshape((-1, n_steps, n_input))
test_label = mnist.test.labels[:test_len]
print "Testing Accuracy:", sess.run(accuracy, feed_dict={x: test_data, y: test_
label})
```

运行结果如下，这个结果虽然比 CNN 结果低些，但也是不错的一个结果。

Testing Accuracy: 0.976562

14.4 分布式 TensorFlow

2016 年 4 月 14 日，Google 发布了分布式 TensorFlow，能够支持在几百台机器上并行训练。分布式的 TensorFlow 由高性能的 gRPC 库作为底层技术支持。

14.4.1 客户端、主节点和工作节点间的关系

分布式架构主要由客户端（Client）和服务端组成，服务端又包括主节点（Master）和工作节点（Worker），这个与 Spark 集群系统有点类似，它们间的交互过程，如图 14-18 所示。

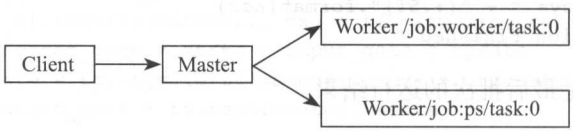


图 14-18 TensorFlow 系统各组件交互

如图 14-18 所示，假设在工作节点存在两个任务：

- /job:ps/task:0: 负责模型参数的存储和更新。
- /job:worker/task:0: 负责模型的训练或推理。

这些组件功能及交互关系如下：

- Client：Client 基于 TensorFlow 的编程接口，构造计算图。目前，TensorFlow 主流支持 Python 和 C++ 等编程接口，TensorFlow 并非立即执行计算。直至建立 Session 会话，并以 Session 为桥梁，建立 Client 与后端运行时的通道。
- Master：在分布式的运行时环境中，Master 根据 Session.run 的 Fetching 参数，从计算图中反向遍历，找到所依赖的最小子图。然后 Master 负责将该子图再次分裂为多个子图片段，以便在不同的进程和设备上运行这些子图片段。最后，Master 将这些图片段派发给 Work Service。随后 Work Service 启动“本地子图”的执行过程。
- Worker：对于每个任务，都存在相应的 Worker Service，它主要负责 3 个方面的职责，包括处理来自 Master 的请求；调度 OP 的 Kernel 实现，执行本地子图；协同任务之间的数据通信。

14.4.2 分布式模式

常用的深度学习训练模型为数据并行化，即 TensorFlow 任务采用相同的训练模型在不同的小批量数据集上进行训练，然后在参数服务器上更新模型的共享参数。TensorFlow 支持同步训练和异步训练两种模型训练方式。

异步训练即 TensorFlow 上每个节点上的任务为独立训练方式，不需要执行协调操作，如图 14-19 所示。

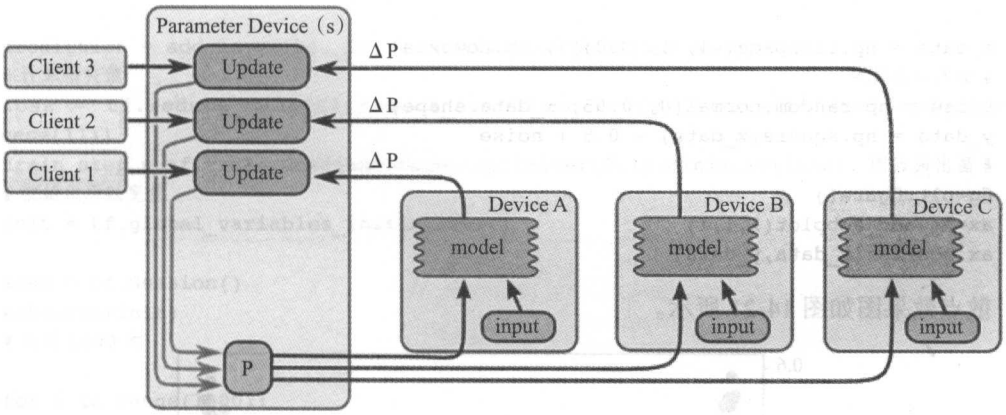


图 14-19　TensorFlow 异步数据并行处理架构

同步训练为 TensorFlow 上每个节点上的任务需要读入共享参数，执行并行化的梯度计算，然后将所有共享参数进行合并，如图 14-20 所示。

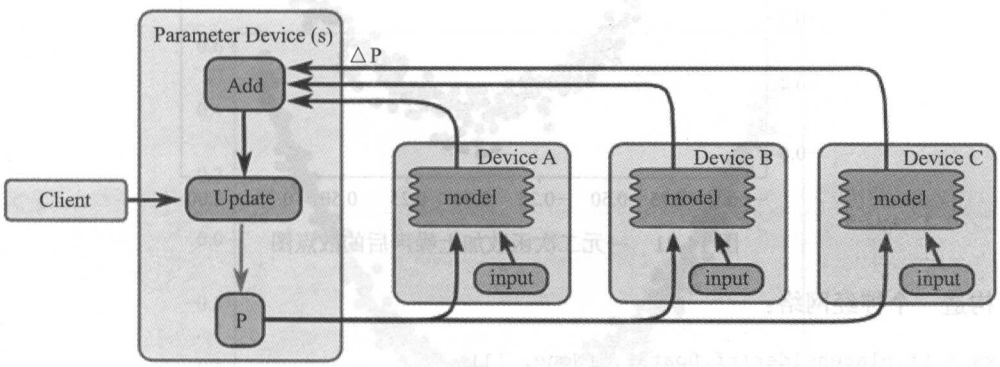


图 14-20　TensorFlow 同步数据并行处理架构

14.4.3　在 Pyspark 集群环境运行 TensorFlow

这节将通过神经网络来模拟一个一元二次方程：$y = x^2 - 0.5$，

TensorFlowOnSpark 的详细配置，请参考 14.4 节。已集群方式启动 PySpark：

```
pyspark --master spark://master:7077 --driver-memory 1G --total-executor-cores 2
```

进入 PySpark 的交换界面：

```
import tensorflow as tf
```

```
import numpy as np
import matplotlib.pyplot as plt

# 构造满足一元二次方程的函数

x_data = np.linspace(-1, 1, 300)[:, np.newaxis]
# 加入一些噪声
noise = np.random.normal(0, 0.05, x_data.shape)
y_data = np.square(x_data) - 0.5 + noise
# 画出散点图
fig=plt.figure()
ax=fig.add_subplot(1,1,1)
ax.scatter(x_data,y_data)
```

散点效果图如图 14-21 所示。

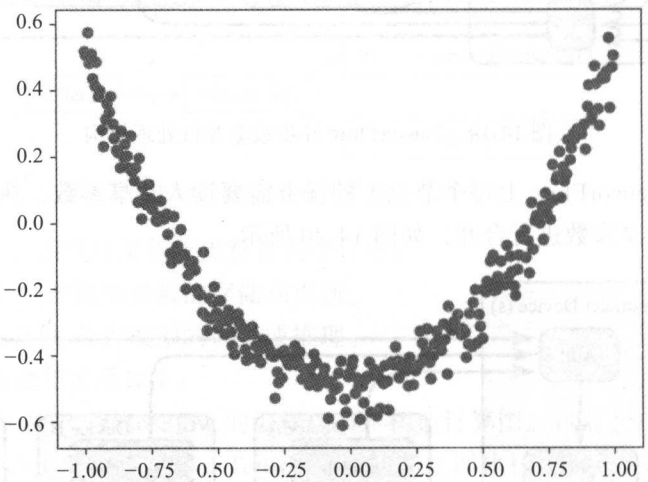


图 14-21 一元二次函数加上噪声后的散点图

构造一个神经网络：

```
xs = tf.placeholder(tf.float32, [None, 1])
ys = tf.placeholder(tf.float32, [None, 1])
# 定义添加层的函数

def add_layer(inputs, in_size, out_size, activation_function=None):
    weights = tf.Variable(tf.random_normal([in_size, out_size]))
    biases = tf.Variable(tf.zeros([1, out_size]) + 0.1)
    Wx_plus_b = tf.matmul(inputs, weights) + biases
    if activation_function is None:
        outputs = Wx_plus_b
    else:
        outputs = activation_function(Wx_plus_b)
    return outputs
# 构造输入层为 1，隐藏层 20 个，输出层为 1 的神经网络
```

```
    h1 = add_layer(xs, 1, 20, activation_function=tf.nn.relu)

    # 构造输出层，隐含层的输出为输出层的输入

    prediction = add_layer(h1, 20, 1, activation_function=None)
    # 计算损失值
    loss = tf.reduce_mean(tf.reduce_sum(tf.square(ys - prediction),reduction_
indices=[1]))
    train_step = tf.train.GradientDescentOptimizer(0.1).minimize(loss)
    # 初始化所以变量
    init = tf.global_variables_initializer()

    sess = tf.Session()
    sess.run(init)
    # 训练 1000 次

    for i in range(1000):
        sess.run(train_step, feed_dict={xs: x_data, ys: y_data})
        if i % 50 == 0:
            print(sess.run(loss, feed_dict={xs: x_data, ys: y_data}))
            prediction_value=sess.run(prediction,feed_dict={xs:x_data})
            lines=ax.plot(x_data,prediction_value,'r',lw=5)
```

模拟结果如图 14-22 所示。

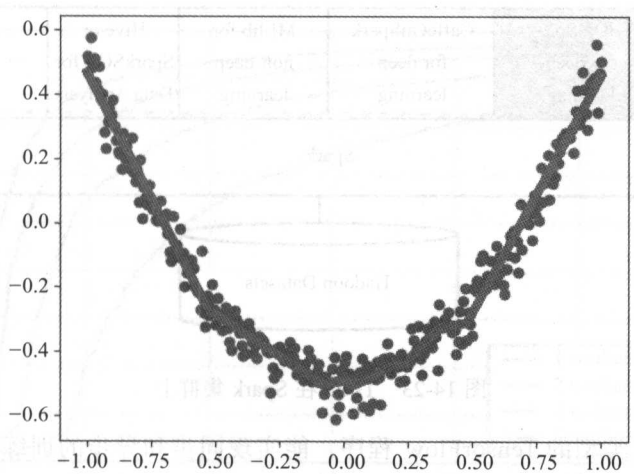


图 14-22　使用 TensorFlow 神经网络模拟的结果

输出结果：

```
1.62758
0.00996406
0.00634915
0.00483868
```

```
0.0043179
0.00399014
0.00368176
0.00337165
0.00309145
0.00284696
0.00267657
0.00255845
0.0024702
0.00240239
0.00235583
0.00232014
0.00229183
0.00226797
0.00224843
```

14.5 TensorFlowOnSpark 架构

TensorFlowOnSpark（TFoS）支持 TensorFlow 在 Spark 和 Hadoop 上的分布式运行。如图 14-23 所示，TFoS 与 SparkSQL、MLlib 以及其他的 Spark 库一起在一个项目或线程中运行。

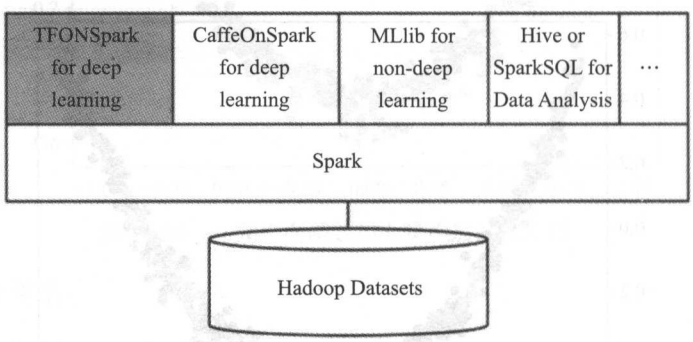


图 14-23 TFoS 在 Spark 集群上

TFoS 支持所有类型的 TensorFlow 程序，能实现同步和异步的训练与推理，并且支持模型和数据的平行处理，以及 TensorFlow 工具（如 TensorBoard）在 Spark 群集上的使用。

任何 TensorFlow 程序都能够通过修改实现在 TFoS 上的运行，通常情况下，只需要修改少于 10 行的 Python 代码。TFoS 支持张量（tensor）在 TensorFlow 处理过程中（计算节点和参数服务节点）信息的直接沟通。过程到过程（Process-to-process）的直接沟通机制使 TFoS 项目很容易在增加的机器上进行扩展。如图 14-24 所示，TFoS 不需要 Spark 驱动器（driver）参与到张量沟通中来，因此也就具备了类似于独立 TensorFlow 群集的扩展能力。

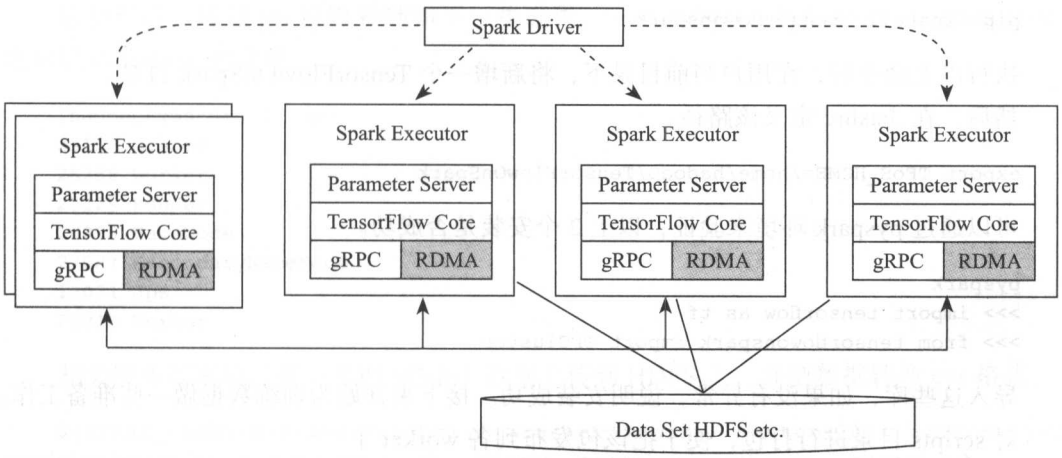


图 14-24　TensorFlowOnSpark 的系统架构

TFoS 曾做过一个图像分类实验，以同一准确度作为评估标准，0.730 的精确度需要单计算节点运行 46 小时得到；双计算节点需要 22.5 个小时，精确度为 0.814；4 计算机点需要 13 小时，精确度为 0.854；8 计算节点需要 7.5 个小时，精确度为 0.879。在 Inception 模型训练上，TFoS 几乎能达到线性扩展，结果鼓舞人心，以上图像分类实验结果如图 14-25 所示。

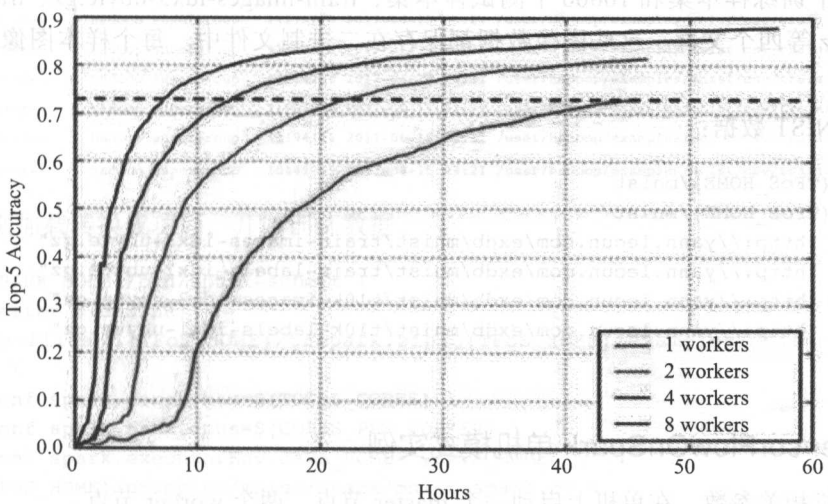


图 14-25　Inception 图像分类网络 TFoS 训练时间与 workers 数量间的关系

14.6　TensorFlowOnSpark 安装

安装 TensorFlowOnSpark，采用 pip 管理工具进行安装，默认安装是 1.0 版本。

```
pip  install  tensorflowonspark
```

执行以上命令后，在用户当前目录下，将新增一个 TensorFlowOnSpark 目录。

然后，在 .bashrc 定义该路径。

```
export TFoS_HOME=/home/hadoop/TensorFlowOnSpark
```

可以通过 pyspark 环境来验证，以上 2 个安装是否成功。

```
pyspark
>>> import tensorflow as tf
>>> from tensorflowonspark import TFCluster
```

导入这些库，如果没有异常，说明安装成功。接下来开始为训练数据做一些准备工作。

对 scripts 目录进行打包，便于把该包发布到各 worker 上

```
cd TensorFlowOnSpark/scripts
zip  -r ../tfspark.zip *
```

14.7 TensorFlowOnSpark 实例

使用 TensorFlowOnSpark 对 MNIST 数据进行预测，MNIST 是一个手写数字数据库，它有 60000 个训练样本集和 10000 个测试样本集，train-images-idx3-ubyte.gz、train-labels-idx1-ubyte.gz 等四个文件。这些图像数据都保存在二进制文件中。每个样本图像的宽高为 28*28。

下载 MNIST 数据：

```
mkdir ${TFoS_HOME}/mnist
pushd ${TFoS_HOME}/mnist
curl -O "http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz"
curl -O "http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz"
popd
```

14.7.1 TensorFlowOnSpark 单机模式实例

设置本机相关参数，在单机上启动一个 master 节点，两个 worker 节点。

```
export MASTER=spark://$(hostname):7077
export SPARK_WORKER_INSTANCES=2
export CORES_PER_WORKER=1
export TOTAL_CORES=$((${CORES_PER_WORKER}*${SPARK_WORKER_INSTANCES}))
${SPARK_HOME}/sbin/start-master.sh; ${SPARK_HOME}/sbin/start-slave.sh -c $CORES_
PER_WORKER -m 2G ${MASTER}
```

启动以后，通过 jps 可以看到如下一些进程。一个 master，两个 worker，namenode 是之前启动 hadoop 的进程。

```
[hadoop@master ~]$ jps
25157 Master
25258 Worker
27229 RunJar
26893 NameNode
27087 SecondaryNameNode
13071 Jps
25215 Worker
```

相关服务起来后，接下来把 MNIST 数据上传到 HDFS 上，并把数据转换 cvs 格式。

```
${SPARK_HOME}/bin/spark-submit --master spark://master:7077 ${TFoS_HOME}/
examples/mnist/mnist_data_setup.py --output /examples/mnist/csv --format csv
```

运行完成后，通过 hadoop fs 命令可以在 HDFS 上看到如下信息：

```
hadoop fs -ls /user/hadoop/examples/mnist/csv/train/images
Found 11 items
-rw-r--r--   1 hadoop supergroup          0 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/_SUCCESS
-rw-r--r--   1 hadoop supergroup    9338236 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00000
-rw-r--r--   1 hadoop supergroup   11231804 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00001
-rw-r--r--   1 hadoop supergroup   11214784 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00002
-rw-r--r--   1 hadoop supergroup   11226100 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00003
-rw-r--r--   1 hadoop supergroup   11212767 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00004
-rw-r--r--   1 hadoop supergroup   11173834 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00005
-rw-r--r--   1 hadoop supergroup   11214285 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00006
-rw-r--r--   1 hadoop supergroup   11201024 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00007
-rw-r--r--   1 hadoop supergroup   11194141 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00008
-rw-r--r--   1 hadoop supergroup   10449019 2017-06-15 23:21 /user/hadoop/examples/mnist/csv/train/images/part-00009
```

数据加载转换成功后，开始训练数据。

```
${SPARK_HOME}/bin/spark-submit \
--master ${MASTER} \
--py-files ${TFoS_HOME}/tfspark.zip,${TFoS_HOME}/examples/mnist/spark/mnist_
dist.py \
--conf spark.cores.max=${TOTAL_CORES} \
--conf spark.task.cpus=${CORES_PER_WORKER} \
--conf spark.executorEnv.JAVA_HOME="$JAVA_HOME" \
${TFoS_HOME}/examples/mnist/spark/mnist_spark.py \
--cluster_size ${SPARK_WORKER_INSTANCES} \
--images examples/mnist/csv/train/images \
--labels examples/mnist/csv/train/labels \
--format csv \
--mode train \
--model mnist_model
```

运行完成后，可以看到如下内容：

```
2017-06-18 05:30:50,072 INFO (MainThread-25741) Feeding training data
2017-06-18 05:32:07,655 INFO (MainThread-25741) Stopping TensorFlow nodes
2017-06-18 05:32:07,883 INFO (MainThread-25741) Shutting down cluster
2017-06-18T05:32:13.346161 ===== Stop
```

如果运行过程中，过程被卡，可以调整 mnist_dist.py 文件中两处代码（在 115,125 行），将 logdir=logdir 改为 logdir=None。

训练完成后，接下来就是用测试集验证模型，并对结果进行预测。

```
${SPARK_HOME}/bin/spark-submit \
--master ${MASTER} \
--py-files ${TFoS_HOME}/tfspark.zip,${TFoS_HOME}/examples/mnist/spark/mnist_
dist.py \
--conf spark.cores.max=${TOTAL_CORES} \
--conf spark.task.cpus=${CORES_PER_WORKER} \
--conf spark.executorEnv.JAVA_HOME="$JAVA_HOME" \
${TFoS_HOME}/examples/mnist/spark/mnist_spark.py \
--cluster_size ${SPARK_WORKER_INSTANCES} \
--images examples/mnist/csv/test/images \
--labels examples/mnist/csv/test/labels \
--mode inference \
--format csv \
--model mnist_model \
--output predictions
```

运行完成以后，在 HDFS 上，就可看到 predictions 目录及相关内容。

```
[hadoop@master spark]$ hadoop fs -ls /user/hadoop/predictions
17/06/20 02:45:57 WARN util.NativeCodeLoader: Unable to load native-hadoop
library for your platform... using builtin-java classes where applicable
Found 11 items
-rw-r--r--   1 hadoop supergroup          0 2017-06-18 14:04 /user/hadoop/predictions/_SUCCESS
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00000
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00001
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00002
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00003
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00004
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00005
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00006
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00007
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00008
-rw-r--r--   1 hadoop supergroup      51000 2017-06-18 14:04 /user/hadoop/predictions/part-00009
```

打开其中一个文件，可以看到预测结果信息。

```
2017-06-18T05:51:42.397905 Label: 5, Prediction: 5
2017-06-18T05:51:42.397923 Label: 9, Prediction: 8
2017-06-18T05:51:42.397941 Label: 7, Prediction: 5
2017-06-18T05:51:42.397958 Label: 3, Prediction: 5
2017-06-18T05:51:42.397976 Label: 4, Prediction: 8
```

```
2017-06-18T05:51:42.397993 Label: 9, Prediction: 8
2017-06-18T05:51:42.398012 Label: 6, Prediction: 5
```

14.7.2　TensorFlowOnSpark 集群模式实例

设置本机相关参数，再以集群方式启动 Spark，一个 master 节点，slave1、slave2 作为两个 worker 节点，各节点资源配置信息如图 14-26 所示。

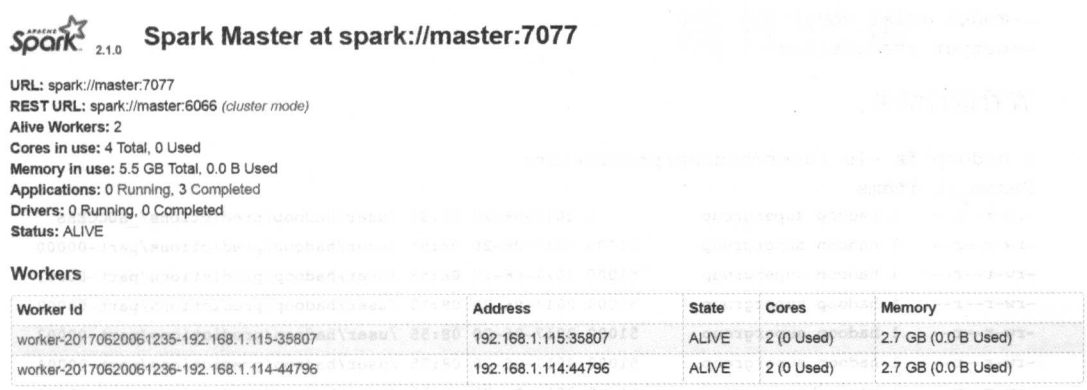


图 14-26　TensorFlowOnSpark 集群模型运行监控图

训练模型：

```
${SPARK_HOME}/bin/spark-submit \
--master spark://master:7077 \
--py-files ${TFoS_HOME}/tfspark.zip,${TFoS_HOME}/examples/mnist/spark/mnist_
dist.py \
--conf spark.cores.max=4 \
--conf spark.task.cpus=2 \
--conf spark.executorEnv.JAVA_HOME="$JAVA_HOME" \
--conf spark.executorEnv.LD_LIBRARY_PATH="${JAVA_HOME}/jre/lib/amd64/server" \
--conf spark.executorEnv.CLASSPATH="$($HADOOP_HOME/bin/hadoop classpath
--glob):${CLASSPATH}" \
${TFoS_HOME}/examples/mnist/spark/mnist_spark.py \
--cluster_size 2 \
--images examples/mnist/csv/train/images \
--labels examples/mnist/csv/train/labels \
--format csv \
--mode train \
--model mnist_model
```

测试模型：

```
${SPARK_HOME}/bin/spark-submit \
--master ${MASTER} \
--py-files ${TFoS_HOME}/tfspark.zip,${TFoS_HOME}/examples/mnist/spark/mnist_
dist.py \
```

```
--conf spark.cores.max=4 \
--conf spark.task.cpus=2 \
--conf spark.executorEnv.JAVA_HOME="$JAVA_HOME" \
${TFoS_HOME}/examples/mnist/spark/mnist_spark.py \
--cluster_size 2 \
--images examples/mnist/csv/test/images \
--labels examples/mnist/csv/test/labels \
--mode inference \
--format csv \
--model mnist_model \
--output predictions
```

查看运行结果：

```
$ hadoop fs -ls /user/hadoop/predictions
Found 11 items
-rw-r--r--   1 hadoop supergroup          0 2017-06-20 08:55 /user/hadoop/predictions/_SUCCESS
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00000
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00001
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00002
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00003
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00004
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00005
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00006
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00007
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00008
-rw-r--r--   1 hadoop supergroup      51000 2017-06-20 08:55 /user/hadoop/predictions/part-00009
```

```
2017-06-20T08:55:06.398866 Label: 9, Prediction: 9
2017-06-20T08:55:06.398877 Label: 9, Prediction: 9
2017-06-20T08:55:06.398888 Label: 5, Prediction: 6
2017-06-20T08:55:06.398899 Label: 5, Prediction: 2
2017-06-20T08:55:06.398911 Label: 1, Prediction: 2
2017-06-20T08:55:06.398922 Label: 5, Prediction: 6
2017-06-20T08:55:06.398933 Label: 6, Prediction: 9
2017-06-20T08:55:06.398945 Label: 0, Prediction: 2
2017-06-20T08:55:06.398956 Label: 3, Prediction: 1
```

运行时各节点报错信息，可以查看 spark/work/app-20170620085449-0003/1 下的

```
-rwxrwxr-x 1 hadoop hadoop  6530 Jun 20 08:54 mnist_dist.py
-rw-rw-r-- 1 hadoop hadoop  4926 Jun 20 08:54 mnist_dist.pyc
-rwxrwxr-x 1 hadoop hadoop  3504 Jun 20 08:54 mnist_spark.py
-rw-rw-r-- 1 hadoop hadoop 14650 Jun 20 08:55 stderr
-rw-rw-r-- 1 hadoop hadoop     0 Jun 20 08:54 stdout
drwxr-xr-x 2 hadoop hadoop  4096 Jun 20 08:55 tensorboard_1
-rwxrwxr-x 1 hadoop hadoop 19016 Jun 20 08:54 tfspark.zip
```

14.8 小结

为了弥补 Spark 机器学习中，缺乏神经网络、深度学习等的不足，这章我们介绍脱胎于 AlphaGo 的深度学习框架 TensorFlow，以基础知识为主，在这个基础上介绍了使用 TensorFlow 的几个实例，最后介绍 TensorFlow 的分布式架构及与 Spark 整合的架构 TensorFlowOnSpark。

附录 A *Appendix A*

线性代数

A.1 向量与空间

A.1.1 向量

顾名思义，向量即一个有方向的量。那量怎么解释呢？——就是把数排成一列吧？比如：

$P=(2,5)^T, q=(3,\sqrt{6},9)^T$，其中 P 就是二维向量，q 就是三维向量。



本书中涉及的向量，都是“列向量”，也就是行向量增加了 T 符号——转置符号。

定义向量的数据结构，就要定义他们的运算了，向量的加法和数量乘法如下：

加法（前提是两者具有相同维数）：

$$\begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 + y_1 \\ \vdots \\ x_n + y_n \end{pmatrix}$$

数乘：

$$c\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} cy_1 \\ \vdots \\ cy_n \end{pmatrix}$$

注意 如下的向量运算并不成立：

$$\begin{pmatrix}x_1\\ \vdots\\ x_n\end{pmatrix}\begin{pmatrix}y_1\\ \vdots\\ y_n\end{pmatrix}=\begin{pmatrix}x_1y_1\\ \vdots\\ x_ny_n\end{pmatrix}\quad\begin{pmatrix}x_1\\ \vdots\\ x_n\end{pmatrix}/\begin{pmatrix}y_1\\ \vdots\\ y_n\end{pmatrix}=\begin{pmatrix}x_1/y_1\\ \vdots\\ x_n/y_n\end{pmatrix}$$

——这个就牵扯到向量的几何意义了。

A.1.2 空间

其实向量的几何意义，就是空间中的矢量：有方向的线段——满足两个条件：①量、②方向。

2 维向量可以在方格纸上描画出来，比如 $(3,6)^T$ 表示横坐标为 3，纵坐标为 6 的 2 维向量。而原点表示横纵坐标均为 0 的零向量。

像这样强调向量的位置的时候，也称为位置向量。

那这个位置向量怎么跟有向线段结合起来的呢？我们可以定义所有的位置向量都以原点为起始点，终点是该向量对应的位置。在用有向线段来解释向量的情况下，就可以通过图形来了解向量的加法和数量乘法。这时，向量加法就变成了线段的连接，数量乘法就变成了线段的伸缩。

A.2 矩阵和行列式

A.2.1 矩阵

既然我们已经引入了向量的概念，接下来要关心的就是对象之间的关系，为了表示这种关系，我们导入矩阵的概念。

那么什么是矩阵？——把数排列成长方形就是矩阵。比如：

$$\begin{pmatrix}2 & 6\\ 3 & 5\end{pmatrix},\begin{pmatrix}1 & \sqrt{2} & 0\\ 4 & 8 & \sqrt{3}\end{pmatrix}$$

以上的矩阵分别是 2×2 和 2×3 的矩阵，同时，2×2 的又叫 2 阶方阵（矩阵的行和列的维数相等，就叫方阵）。

矩阵 A 中第 i 行第 j 列的值，称为该矩阵 A 的 (i,j) 元素，比如上例中第二个矩阵（2，2）为 8，（2，1）为 4，当提到 (i,j) 元素时，是先行后列的顺序，按照下式中的规则书写的时候，变量的下标一般也是先行后列的顺序。

矩阵的加法和数量乘法和向量类似，加法就是对应元素相加（前提是行列一致），乘法就是数量乘以矩阵中的每个元素，如下所示：

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}+\begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & & \vdots \\ b_{m1} & \cdots & b_{mn} \end{pmatrix}=\begin{pmatrix} a_{11}+b_{11} & \cdots & a_{1n}+b_{1n} \\ \vdots & & \vdots \\ a_{m1}+b_{m1} & \cdots & a_{mn}+b_{mn} \end{pmatrix}$$

$$c\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}=\begin{pmatrix} ca_{11} & \cdots & ca_{1n} \\ \vdots & & \vdots \\ ca_{m1} & \cdots & ca_{mn} \end{pmatrix}$$

下面的运算规则，和向量的运算是完全一样的：

$$-\mathrm{A}=(-1)\mathrm{A}$$

$$\mathrm{A}-\mathrm{B}=\mathrm{A}+(-\mathrm{B})$$

$$2\mathrm{A}+3\mathrm{B}=(2\mathrm{A})+(3\mathrm{B})$$

但是如果是两个矩阵相乘，情况就完全不同了。

比如，对于 m × n 维矩阵与 n 维向量（n × 1 的矩阵），有：

$$\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}\begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}=\begin{pmatrix} a_{11}y_1+\cdots+a_{1n}y_n \\ \vdots \\ a_{m1}y_1+\cdots+a_{mn}y_n \end{pmatrix}$$

关于矩阵和向量乘积，有几点需要注意：

- 矩阵与向量的乘积是向量。
- 矩阵的列数（宽度）为输入的向量维数，行数（高度）为输出的向量维数。

总而言之，矩阵就是一种表示平直关系的便利手段。

关于这种平直关系怎么体现，我们举个例子来说明：

现有一笼，其中有鸡 $X_{鸡}$ 只，兔 $X_{兔}$ 只，笼中头的总数 $Y_{头}$ 和脚的总数 $Y_{脚}$，可得：

$$Y_{头}=a_{头鸡}X_{鸡}+a_{头鸡}X_{兔}=X_{鸡}+X_{兔}$$

$$Y_{脚}=a_{脚鸡}X_{鸡}+a_{脚鸡}X_{兔}=2X_{鸡}+4X_{兔}$$

其中鸡和兔都是一个头，所以 X 前面的系数是 1。而 $a_{脚鸡}$ 表示一只鸡两只脚，$a_{脚兔}$ 表示一只兔子四条腿。以上的方程如果用矩阵表示就是这样子的：

$$\begin{pmatrix} Y_{头} \\ Y_{脚} \end{pmatrix}=\begin{pmatrix} a_{头鸡} & a_{头兔} \\ a_{脚鸡} & a_{脚兔} \end{pmatrix}\begin{pmatrix} X_{鸡} \\ X_{兔} \end{pmatrix}=\begin{pmatrix} 1 & 1 \\ 2 & 4 \end{pmatrix}*\begin{pmatrix} X_{鸡} \\ X_{兔} \end{pmatrix}$$

矩阵给出了头和脚之间的函数对应关系。也就是说，如果我们找到了某个对应矩阵 A，就能找到 Y 与 X（这里是头和脚）的对应关系。说白了，矩阵就是一种映射。

A.2.2 行列式

我们回到上面，来看鸡和兔子的头和脚的对应关系：

$$\begin{pmatrix} Y_{头} \\ Y_{脚} \end{pmatrix}=\begin{pmatrix} a_{头鸡} & a_{头兔} \\ a_{脚鸡} & a_{脚兔} \end{pmatrix}\begin{pmatrix} X_{鸡} \\ X_{兔} \end{pmatrix}=\begin{pmatrix} 1 & 1 \\ 2 & 4 \end{pmatrix}*\begin{pmatrix} X_{鸡} \\ X_{兔} \end{pmatrix}$$

我们假设头和脚对应的向量跟坐标轴围成一个体积大于零的立方体，而鸡和兔子的只数表示其对应的向量跟坐标轴围成的另外一个体积大于零的立方体。而中间矩阵表示的是两者的映射，而对它取行列式的话，就是指两个体积之间的倍数关系——体积的扩大率。

行列式有如下性质：

性质 1 行列互换，行列式不变。

性质 1 表明，在行列式中行与列的地位是对称的，因而凡是有关行的性质，对列也同样成立。例如由（8）即得下三角形的行列式

性质 2 如果行列式中一行为零，那么行列式为零。

这就是说，一行的公因子可以提出去，或者说以一个数乘行列式的一行相当于用这个数乘此行列式。于是，就有如果行列式中一行为零，那么行列式为零。

性质 3

这就是说，如果某一行是两组数的和，那么这个行列式就等于两个行列式的和，而这两个行列式除这一行以外全与原来行列式的对应的行一样。

性质 3 显然可以推广到某一行为多组数的和的情形。

性质 4 如果行列式中有两行相同，那么行列式为零。所谓两行相同就是说两行的对应元素都相等。

性质 5 如果行列式中两行成比例，那么行列式为零。

性质 6 把一行的倍数加到另一行，行列式不变。

性质 7 对换行列式中两行的位置，行列式反号。

行列式和矩阵将我们研究的机器学习问题从低维延伸到高维。使得很多本来在低维无解的机器学习方法，在高维能找到答案（比如 SVM）。

A.3 特征值与特征向量

一般而言，对于方阵 A，满足：

$AP = \lambda p, p \neq 0$ 的数 λ 和非零向量 p 分别称为特征值和特征向量。

关于特征值和特征向量有以下两个性质：

1 矩阵不同的特征值对应的特征向量线性无关。

2 如果 $n \times n$ 矩阵 A 的特征多项式有不同的根，则 A 有 n 个线性无关的特征向量。

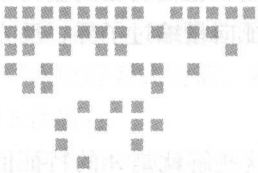
从几何学意义上讲，任意一个矩阵乘法对应了一个变换，是把任意一个向量变成另一个方向或长度都大不相同的新向量。在这个变换的过程中，原向量主要发生旋转、伸缩的变化。如果矩阵对某一个向量或某些向量只发生伸缩变换，不对这些向量产生旋转的效果，那么这些向量就称为这个矩阵的特征向量，伸缩的比例就是特征值。

实际上，上述的一段话既讲了矩阵变换特征值及特征向量的几何意义（图形变换）也讲了其物理含义。物理的含义就是运动的图景：特征向量在一个矩阵的作用下作伸缩运动，

伸缩的幅度由特征值确定。特征值大于1，所有属于此特征值的特征向量身形暴长；特征值大于0小于1，特征向量身形猛缩；特征值小于0，特征向量缩过了界，反方向到0点那边去了。

特征值和特征向量的计算：

对于任意方阵A，首先求出方程$|\lambda E-A|=0$的解，这些解就是A的特征值，再将其分别代入方程$(\lambda E-A)p=0$中，求得它们所对应的基础解系，则对于某一个λ，以它所对应的基础解系为基形成的线性空间中的任意一个向量，均为λ所对应的特征向量。



Appendi B 附录 B

概率统计

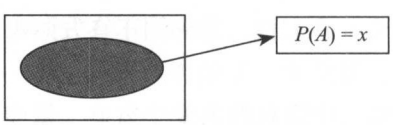
B.1 随机变量与概率分布

B.1.1 概念

随机变量在英文中称为 random variable，简写为 r.v.。通俗来讲，它是一种会随机改变的不确定量。那既然不确定，我们又是如何去度量它的呢？

如果我们单纯表面地看这个问题：随机变量就是某一事件发生的概率，记为 $P(A) = x$，x 会随着时间和不同场景而变化。那这样的话，我们很难从直观上去理解这个概率问题。那有什么办法呢？

有，我们可以把概率当成一种面积去理解，所有的概率问题都可以表示成一个小于一大于 0 的面积的大小。这样就确定概率了。而随机的部分就是这个面积的形状，而它的值，就是唯一确定的了：



而这个面积是怎么算出来的呢？既然我们已经知道了随机变量的概率，那这个面积，其实就是该概率下的概率分布。

概率分布的概念比随机变量更为宽泛，它只考虑面积，不考虑其他的。在这里我们直接把概率分布称作分布。比如掷骰子中骰子点数出现的概率就是一种分布：

点数	正常的骰子出现该点数的概率	老千的骰子出现该点数的概率
1	1/6	0.4
2	1/6	0.1
3	1/6	0.1
4	1/6	0.1
5	1/6	0.1
6	1/6	0.2

在以上的例子中，掷骰子的两种情况（正常的和老千的），就是两种不同的随机变量，而在某一个点数下得到的概率值，就是该随机变量的概率分布（或者说分布）。

在日常生活中，我们有更一般的说法：对于更为一般的随机变量，我们有如下的公式计算概率分布：

$$P(x=k)=x(w)=k \text{ 时区域 } w \text{ 的面积}$$

这里，我们需要注意以下两点：

1）所有的概率分布下的每个点的概率值介于 0 到 1 之间。

2）同一概率分布下的概率之和为 1。

B.1.2 表示方法

原则上，随机变量的表示一般以大写为主，比如 $X(w)=a$，那么 X 就是随机变量，而 a 就是结果。这点跟函数的映射很相似，如果我们再加入概率分布，就会有：$P(x=k)=a$。——两个等号可能感觉有点拖沓。为了简单明了又能说明问题，我们通常用 $P(k)$ 表示随机变量 x 在 k 处的概率值。

那么，如果有多个随机变量呢？最通常的做法就是写成 $P(x=3), P(y=2)$。而更一般的，我们也可以写成 $Px(3)$ 或者 $Py(2)$。

B.2 条件概率与贝叶斯

B.2.1 前言

在说这个之前，我们先来搞清楚什么是联合概率和边缘概率。

假设有两个随机变量 X 与 Y，此时 $P(X=a,Y=b)$ 用于表示 $X=a$ 且 $Y=b$ 的概率，像这样的包含多个条件且所有条件同时成立的概率称为联合概率。请注意：联合概率不是其中某个条件成立的概率，而是所有条件同时成立的概率。而边缘概率是指仅仅像 $P(X=a)$ 或者 $P(Y=b)$ 这类仅仅与一个随机变量有关的概率称为边缘概率。

联合概率与边缘概率额关系如下：

$$P(x=a)=\sum_{b}P(x=a,y=b)$$

$$P(y=b)=\sum_{a}P(x=a,y=b)$$

如果用面积表示的话，X 和 Y 两者相交的部分就是联合概率了，而 X 与 Y 的单独部分，就是他们的边缘概率。

B.2.2 公式推导

说了那么多，那什么叫条件概率呢？

在实际生活中，许多有价值的变量都能以条件概率这一概念来表述。我们以扑克牌为例：

一副扑克牌一共 54 张，去掉 2 个王，剩下 52 张，其中 1-10 一共 40 张。而红色的牌一共 26 张，而数字牌一共 40 张，所以就有：

$$P(x=\text{数字牌})=40/52$$

$$P(x=\text{红色})=26/52=1/2$$

那既是红色又是数字的概率就是 $P(X=\text{数字牌}, y=\text{红色})=20/52$ 了。

好了，那我如果抽到一张牌是红色的，它是数字牌的概率是多少？

这里就要引入条件概率了，我们可以把这个问题想成：当“红色牌”成立的时候，“数字牌”发生的概率是多少？简单运算之后可得：

$$P(x=\text{数字牌}\mid y=\text{红色})=\frac{P(X=\text{数字牌},y=\text{红色})}{P(y=\text{红色})}=10/52$$

所以这就是条件概率的计算方法。

我们把它一般化了就得到：

$$P(x=a\mid y=b)=\frac{P(x=a,y=b)}{P(y=b)}$$

那贝叶斯公式又是什么？说这个之前，我们先来看看这个问题：

已知 $x=a,b,c;y=\mathrm{d}$

我们又知道：$P(x=a),P(x=b),P(x=c)$ 和

$P(y=d|x=a),P(y=d|x=b),P(y=d|x=c)$

那么如果我知道 $y=d$，那 $x=a$ 的概率有多少？

然后根据条件概率，我们可以推得：

$$P(x=a\mid y=d)=\frac{P(x=a,y=d)}{p(y=d)}$$

发现分子很好求，根据条件概率可得：$P(x=a,y=d)=P(y=d|x=a)\cdot P(x=a)$

那分母呢？别忘了，$y=d$ 中有三种情况：x=a,$x=b$ 和 $x=c$。于是我们依然可以用条件概

率求得分母：

$$P(y=d)=P(y=d\mid x=a)\cdot P(x=a)+P(y=d\mid x=b)\cdot P(x=b)+P(y=d\mid x=c)\cdot P(x=c)$$

两者相除就得到贝叶斯公式：

$$P(x=a\mid y=d)=\frac{P(y=d\mid x=a)\cdot P(x=a)}{P(y=d\mid x=a)\cdot P(x=a)+P(y=d\mid x=b)\cdot P(x=b)+P(y=d\mid x=c)\cdot P(x=c)}$$

B.3 协方差与多元正态分布

B.3.1 协方差

本节将讨论两个随机变量 X，Y 的协方差（covariance），$\mathrm{Cov}(X,Y)$，协方差和方差的表达类似，不过含义有所不同。其中 X，Y 都是随机变量，可取任意大小的值。为了构造协方差，我们规定：

1）协方差为正表示一方增大，另一方也倾向于增大。

2）协方差为负表示一方增大，另一方倾向于减小。

3）协方差为零：X，Y 两者独立。

基于上面的规定，我们得出协方差的结果：$\mathrm{Cov}[X,Y]=E[(X-\mu)(Y-v)]$，不难看出，如果 $X=Y$，$u=v$，协方差就变成了：$E[(X-\mu)^2]$——这个就是方差公式。

针对协方差公式的特性，我们可以从概率上讨论得到更精确的定义：

当协方差的值为正时：

- 如果一方的取值大于期望值，另外一方取值大于期望值的概率也将更大。
- 如果一方的取值小于期望值，另外一方取值小于期望值的概率也将更大。
- 当协方差的值为负时：
- 如果一方的取值大于期望值，另外一方取值小于期望值的概率也将更大。
- 如果一方的取值小于期望值，另外一方取值大于期望值的概率也将更大。

协方差矩阵：

假如有 n 个随机变量，他们之间的协方差系数是怎么表示的呢？这里就要用到协方差矩阵了，比如有随机变量 X_1，X_2，X_3，他们之间的协方差系数如下：

	x_1	x_2	x_3
x_1	$\mathrm{Var}(x_1)$	$\mathrm{Cov}(x_1,x_2)$	$\mathrm{Cov}(x_1,x_3)$
x_2	$\mathrm{Cov}(x_2,x_1)$	$\mathrm{Var}(x_2)$	$\mathrm{Cov}(x_2,x_3)$
x_3	$\mathrm{Cov}(x_3,x_1)$	$\mathrm{Cov}(x_3,x_2)$	$\mathrm{Var}(x_3)$

也就是说：对于任意随机变量 X_i，X_j，我们都有：

$$Cov=(X_i,X_j)=\begin{cases}\text{var}(X_i) & i=j\\ Cov(X_i,X_j) & i\neq j\end{cases}$$

协方差是我们研究不同随机变量之间关系的一个桥梁，同时我们也可以通过求协方差，来研究变量与变量之间的相互关系。

B.3.2 多元正态分布

我们接下来讨论多元正态分布的问题：

顾名思义，多元正态分布是含有多个变量的正态分布，正态分布是一种极为重要的基本分布类型，多元正态分布同样具有重要意义。基于以下两个特征，多元正态分布的应用十分广泛。

多元正态分布的表达式易于处理，且理论推导的结果较为简洁

现实生活中很多问题都可以用多元正态分布解决。

多元正态分布的类型：

1. 多元标准正态分布

我们先定义单个标准正态分布的函数：

$$Z=(Z_1,Z_2\cdots Z_n),g(z)=c\cdot\exp(-\frac{z^2}{2})$$

然后求得联合概率密度：

$$f(Z)=c\cdot\exp(-\frac{z_1^2}{2})\cdot c\cdot\exp(-\frac{z_2^2}{2})\cdots c\cdot\exp(-\frac{z_n^2}{2})=d\ \exp(-\frac{\|z\|^2}{2})$$

其中 $||Z||$ 表示 Z 的模长，d 可以由概率和为 1 唯一确定，

显然，联合概率密度函数的横截面是一个圆（或者说等值面是一个球面或者超球面）。

而多元标准正态分布的方差和数学期望可以由概率密度函数唯一确定。

2. 多元一般正态分布

我们将之推广到一般情况，假设一组符合多元标准正态分布的随机变量的数学期望为 E[Z]，方差是 V[Z]. 我们假设对该标准正态分布进行变换：设 $X=aZ+b$。于是多元一般的正态分布的数学期望和方差如下可得：

$$\text{E}[X]=a\text{E}[Z]+b$$

$$\text{V}[X]=a^2\text{V}[Z]$$

B.4 估计与检验

在完成数据采集之后，我们自然一方面希望统计其中某些值所占的比率——这就是描

述统计所做的事情；另一方面，我们希望能够识别两个数据集的差异——这就是推断统计所做的事情。描述统计是数据分析的最基本的步骤，一般来说我们通过计算数学期望和方差就能知道某个数据集的情况，而如果这个数据集的方差和数学期望很难求得，我们就要通过估计的方法，根据样本的数学期望和方差来大致预测下总体的数学期望和方差。

B.4.1 估计方法介绍——均值估计

估计均值的方法有点估计和区间估计，我们这边就先介绍点估计，假设 X_1，$X_2...X_n$ 是某个正态分布总体的抽样样本。

那我们可以得到这个正态分布总体的点估计值：

$$\begin{cases} u=(x_1+x_2+\cdots+x_n)/n \\ u_2=x_1,x_2\cdots,x_n\text{的中位数} \end{cases}$$ ——如何判断两个估计量谁更精确，我们可以通过最小化损失函数来求得最佳的点估计的值，判断如下：

$E[(u-u')^2]$ 其中 u 是总体的数学期望，u' 是每一次抽样得到的样本的数学期望

此外，除了点估计和区间估计，从方法上我们还有无偏估计，极大似然估计和贝叶斯估计。总之，估计的核心就是通过样本去估计总体的值。

B.4.2 检验理论：

在说检验之前，我们先来理解两个概念：虚无假设与对立假设。虚无假设又称零假设，是我们试图驳斥的主张；对立假设又称备择假设，是我们试图肯定的主张。我们用一场球赛的获胜与否作为假设的例子来说明：

H0：甲队获胜（零假设）。HA：乙队获胜（备择假设）

这里有个问题要请大家注意，其实原则上，我们希望乙队获胜——也就是备择假设成立。而如果只是乙队获胜一个假设，不足以支持我们，所以我们希望有参照组，也就是甲队获胜。

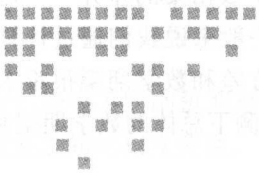
然而比赛的胜负随机性很大，所以我们不能直接确定哪个假设是正确的。因此我们进一步降低了要求，我们用一个比例来度量 HA 为真的概率的大小，这样我们就能知道 H0 是否是错的，从而肯定 HA。

这个比例的大小称为 P 值，而 P 值的阈值称为拒绝域 α，根据这个比例和拒绝域的关系，我们有（不管是否拒绝，HA 为真的概率都为 $1-\alpha$）：

$P<\alpha$——拒绝 H0（reject）

$P>\alpha$——无法拒绝 H0（accept）

这里有个注意点要说明。拒绝零假设 H0 其实是在主张 HA 的正确性。反之，如果上述主张与实际情况不太相符，则无法拒绝 H0。也就是说，即使 H0 成立，我们也有很大的概率得到现在的数据，因此不能断言 H0 无法成立。



Appendi C 附录 C

Scala 基础

C.1 Scala 简介

C.1.1 Scala 概述

对于初学者来说，学一种语言或工具，首先关心的学习该语言或工具的意义、特点及作用等。

Scala 是一种有趣的语言。它一方面吸收继承了多种语言中的优秀特性，一方面又没有抛弃 Java 这个强大的平台，它运行在 Java 虚拟机（Java Virtual Machine）之上，轻松实现和丰富的 Java 类库互联互通。它既支持面向对象的编程方式，又支持函数式编程。它写出的程序像动态语言一样简洁，但事实上它确是严格意义上的静态语言。Scala 将过去几十年计算机语言发展历史中的精萃集于一身，化繁为简，为程序员们提供了一种新的选择。

目前大数据运算架构 Spark 就是用 Scala 编写的，在 Spark 中把 scala 的分布式、高并发、简洁等特性发挥得淋漓尽致。

C.1.2 Scala 简介

Scala 是 Scalable Language 的简写，是联邦理工学院洛桑（EPFL）的 Martin Odersky 于 2001 年基于 Funnel 的工作开始设计的。Funnel 是把函数式编程思想和 Petri 网相结合的一种编程语言。Odersky 先前的工作是 Generic Java 和 javac（Sun Java 编译器）。

Java 平台的 Scala 于 2003 年年底 /2004 年年初发布。.NET 平台的 Scala 发布于 2004 年

6 月。该语言第二个版本 v2.0，发布于 2006 年 3 月。截至目前，最新版本是版本 2.11。

Scala 主要特性：

- 面向对象：Scala 是一种纯面向对象的语言，每个值都是对象。
- 函数式编程：Scala 也是一种函数式语言，其函数也能当成值来使用。Scala 提供了轻量级的语法用以定义匿名函数，惰性求值，支持高阶函数，允许嵌套多层函数，并支持柯里化。Scala 的 case class 及其内置的模式匹配相当于函数式编程语言中常用的代数类型。
- 静态类型：Scala 具备类型系统，通过编译时检查，保证代码的安全性和一致性。
- 扩展性：Scala 提供了许多独特的语言机制，可以以库的形式轻易无缝添加新的语言结构。
- 并发性：Scala 使用 Actor 作为其并发模型，Actor 是类似线程的实体，通过邮箱发收消息。Actor 可以复用线程，因此可以在程序中可以使用数百万个 Actor，而线程只能创建数千个。在 2.10 之后的版本中，使用 Akka 作为其默认 Actor 实现。

C.1.3 Scala 与 Java、Python

1. Scala 与 Java 异同

1）两者都是基于基于 JVM 及 JAVA 的生态系统。

2）scala 不需要分号结尾，return 可以省略。

2. Scala 与 python 异同

1）两者都是面向对象，也拥有一个 REPL。

2）函数定义都以 def 开头。

3）Scala 不像 Python 一样通过代码缩进来表示代码的层次关系，而是和通常的语言一样使用 {} 来表示代码的层级。

4）scala 缺少 python 那样丰富的数据处理，python 不是为大数据设计的，scala 可以说是大数据导向的。

C.1.4 安装配置

Scala 基于 java 之上，大量使用 java 的类库和变量，在使用 Scala 之前必须先安装 Java。

第 1 步：安装配置 Java。

下载安装 JDK 1.6 以上版本，设置 JAVA_HOME 环境变量及 JDK 的 bin 目录。

在 /etc/profile 配置文件中：

```
# 追加如下内容
export JAVA_HOME=/usr/local/java
```

```
export PATH=$PATH:$JAVA_HOME/bin
```

我们可以使用以下命令查看是否安装了 Java：

```
[hadoop@master ~]$ java -version
openjdk version "1.8.0_91"
OpenJDK Runtime Environment (build 1.8.0_91-b14)
OpenJDK 64-Bit Server VM (build 25.91-b14, mixed mode)
[hadoop@master ~]$ javac -version
javac 1.8.0_60
```

第 2 步：安装配置 Scala。

可以从 Scala 官网地址 http://www.scala-lang.org/downloads 下载 Scala 二进制包。

解包到指定目录下：

```
tar  zxvf scala-2.11.8.tgz -C /usr/local/
### 重命名
mv scala-2.11.8  scala
```

在 /etc/profile 配置文件中：

```
# 追加如下内容
export SCALA_HOME=/scala
export PATH=$PATH:$SCALA_HOME/bin
```

然后 source /etc/profile 是该配置文件立即生效

运行 scala 命令，查看 scala 安装是否成功。

```
[hadoop@master ~]$ scala
Welcome to Scala 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_60).
Type in expressions for evaluation. Or try :help.

scala>
```

终端出现了 scala> 提示符，和 Python 的 REPL 一样，Scala 也有一个 REPL。现在我们可以开始写一些语句或编写一些代码，进行体验或测试。

```
scala> 4+5
res3: Int = 9       //res3 是解释器自动生成的变量名，后面可直接使用该变量
scala> res3*10
res4: Int = 90

scala> res4+100
res5: Int = 190
```

scala 是一种伸缩性很强的语句，尤其是适合于函数式编程，可以像 Python 一样作为脚本语言来执行，如下例所示。

```
[hadoop@master ~]$ cat test.scala
#!/bin/env scala
println("hello " +args(0)+"!")
[hadoop@master ~]$ ./test.scala 高峰
hello 高峰!
```



以上是安装在 centos 上，如果安装在 ubuntu 上，可采用 dpkg -i file.deb 的方式。

C.2　基础知识

C.2.1　常量与变量

Scala 中常用以 val 定义，变量以 var 定义。val(value) 表示常量，为不可变的，不可修改，但在 REPL 可以重新赋值。var(variable) 表示可变的，可以重新赋值或修改。定义变量时，可不显式指定数据类型，Scala 有很强类型自动推导功能，Scala 具有动态语言似的编写代码，但又有静态语言的编译时检查。

```
scala> val a=10
a: Int = 10
scala> val b=10.5
b: Double = 10.5

scala> b=12.5    //b是常量类型，不可修改，否则报错。
<console>:8: error: reassignment to val
       b=12.5
       ^

scala> var c=80
c: Int = 80

scala> var d:Int=100
d: Int = 100

scala> d=200
d: Int = 200
```

C.2.2　基本类型

Scala 中基础数据类型有：Byte、Short、Int、Long、Float、Double，Boolean，Char、String。和 Java 不同的是，Scala 中没在区分原生类型和装箱类型，如：int 和 Integer。它统一抽象成 Int 类型，这样在 Scala 中所有类型都是对象了。

Scala 中单引号和双引号包裹是有区别的，单引号用于字符，双引号用于字符串，而且双引号中特殊字符，需用转移字符“\”)。此外，还有多行字面量（如 3 个引号）和字符串内插等特性。

Scala 基于 JVM 平台，默认使用 unicode，所以变量名是可以直接用中文的。而在 Scala 中，中文也是直接显示的，不像 Python2 一样会输出成 unicdoe 编码形式：\u**。以

下通过实例说明。

```
scala> val s='h'
s: Char = h

scala> var s1="hello world!"
s1: String = hello world!

scala> var s1="hello \nworld!"
s1: String =
hello
world!
scala> val d="""hello \nworld"""  // 多行字面量，内部字符都视为原始字符
d: String = hello \nworld
scala> val name="scala"
name: String = scala

scala> println(s"the tool is  $name")   // 字符串前加 s，可以内插字符串
the tool is  scala
```

3 个双引号在 Scala 中也有特殊含义，它代表被包裹的内容是原始字符串，可以不需要字符转码。这一特性在定义正则表达式时很有优势。

C.2.3 控制语句

Scala 内置的控制语言有 if、while、for、match case 四大主要控制语句。Scala 的控制语句都是表达式是有返回值的。

1. if 语句格式

```
if (Boolean express)  express
if (Boolean express )  express  else  express
```

示例：

```
scala> val a=10;val b=15
a: Int = 10
b: Int = 15

scala> val c=if ( a>b) a else b
c: Int = 15
```

2. while 语句格式

```
while (Boolean express) {
statement
}
```

示例：

```
scala> var x=2;while (x>0) {
     | x-=1
     | println(x)
     | }
1
0
x: Int = 0
```

3. for 语句格式

```
for (identifies <- iterator)[yield] [express]
```

关键字 yield 是可选的，如果表达式指定这个关键字，所有表达式的值将作为一个集合返回；如果没有指定这个关键字，则调用表达式，但不能访问其返回值。

示例：

```
scala> val list=List(1,2,3)
list: List[Int] = List(1, 2, 3)

scala> for(i <-list) println(i)
1
2
3
scala> for (j <- 1 to 5 ) yield j*2
res4: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)

scala> var list2=for (j <- 1 to 5 ) yield j*2
list2: scala.collection.immutable.IndexedSeq[Int] = Vector(2, 4, 6, 8, 10)
```

Python 有个列表推导式，在 Scala 中也有类似功能，而且更强大，作用库不限于列表，在整个集合（如 Seq、Map、Set、List、Array 等）都适合，当然，在 Scala 中这种方式叫迭代器守卫（iterator guard）。其格式如下：

```
for (identifies <- iterator  if Boolean express )[yield] [express]
```

示例：

```
scala> var list2=for (j <- 1 to 5 if j%2==0) yield j*2
list2: scala.collection.immutable.IndexedSeq[Int] = Vector(4, 8)

scala> val s1="python,ipython,pandas,scala"
s1: String = python,ipython,pandas,scala
scala> for (i <- s1.split(",") if i.size>5) println(i)
python
```

```
ipython
pandas
```

4. match case 语句格式

```
express match{
case pattern_match => express
[case statement]
}
```

Scala 中的 match case 称为模式匹配，它是默认 break 的，只要其中一个 case 语句匹配，就终止之后的所以比较。且对应 case 语句的表达式值将作为整个 match case 表达式的值返回。

示例：

```
scala> val tools="scala"
tools: String = scala

scala> val t1=tools match{
    | case "python"=>"python"
    | case "pandas"=>"python"
    | case "ipython"=>"python"
    | case "scala"=>"scala"
    | case _ =>"others"      // 下划线表示通配符
    | }
t1: String = scala
```

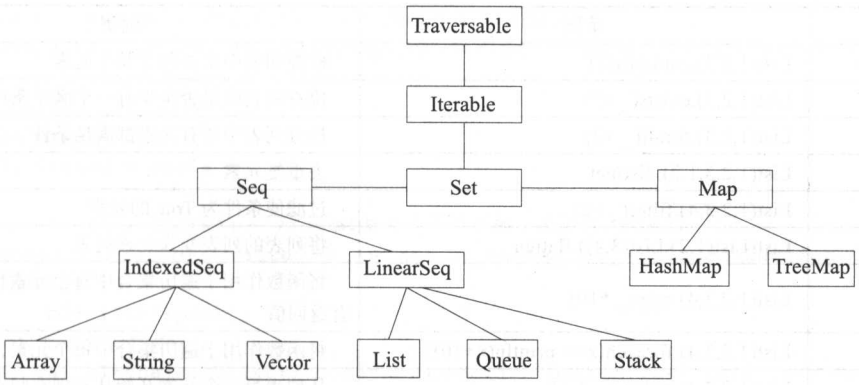
这种多个条件对应一个结果的情况，可以用竖线（|）来表示，这样结构更简洁。

示例：

```
scala> val t1=tools match{
    | case "python"|"pandas"|"ipython" =>"python"
    | case "scala"=>"scala"
    | case _ =>"others"
    | }
t1: String = scala
```

C.3 常用集合

在 Python 中，常用集合类型有：list、tuple、set、dict 等。Scala 中对应的有：Array 、List、Tuple、Set、Map 等。Scala 中这些关键字的第一个字母需大写。Scala 中各集合间的层次结构如下：



C.3.1 数组

数组 Array，下标从 0 开始，通过 () 来引用，成员可修改。

```
scala> val arr1=Array("python","java","scala")
arr1: Array[String] = Array(python, java, scala)

scala> arr1(1)="spark"

scala> arr1
res12: Array[String] = Array(python, spark, scala)
```

C.3.2 列表

列表 List，下标从 0 开始，通过 () 来引用，成员可读，但不可修改，List 是 Iterable 的一个子类型。

```
scala> val list1=List(10,20,5)
list1: List[Int] = List(10, 20, 5)

scala> list1(2)
res13: Int = 5

scala> list1(1)=30 // 成员不可修改，否则报错
<console>:13: error: value update is not a member of List[Int]
       list1(1)=30
```

列表还可通过 :: 操作符，将一个元素和列表连接起来，并把元素放在列表的开头。

```
scala> 1::list1
res16: List[Int] = List(1, 10, 20, 5)
```

列表常用方法：

方法	示例	说明
contains	List(1,2,3).contains(2)	检查列表中是否包含某个元素
exits	List(1,2,3).exists(_ >2)	检查列表中是否至少有一个满足条件的元素。
forall	List(1,2,3).forall(_ >2)	检查列表中所有元素都满足条件。
distinct	List(1,2,3,1,2).distinct	去重复元素
filter	List(1,2,3,4).filter(_ >2)	过滤使条件为 True 的元素
flatten	List(List(1,2),List(3,4)).flatten	将列表的列表变为元素列表
map	List(1,2,3,4) map(_ *10)	将函数作用于遍历集合中每个元素得到新列表，有返回值
foreach	List(1,2,3,4).foreach(x=> println(x+10))	将函数作用于遍历集合中每个元素，无返回值
reduce	List(1,2,3,4) reduce(_ + _)	从列表第一个元素开始从左到右归约处理
sortBy	List(“abc”,” cd”,” efg”).sortBy(_.size)	根据函数返回值进行排序
take	List(1,2,3,4).take(2)	取列表前 n 个元素
size	List(“abc”,” cd”,” efg”).size	统计列表元素个数
zip	List(1,2)zip List(3,4)	将两个列表合并为一个元组列表，列表对应索引组成一个元组

C.3.3 元组

与 Python 中类似，Scala 也采用小括号来定义元组 Tuple，下标从 1 开始，通过 _ 下标来访问成员，不能通过 () 访问成员，这点与数组、列表不同，其成员可以是不同类型，元组最多只支持 22 个元素。

```
scala> val t1=(2,5,1,4)
t1: (Int, Int, Int, Int) = (2,5,1,4)
scala> t1._2
res42: Int = 5
```

C.3.4 集合

Set 是一个不重复且无序的集合，初始化一个集合需要使用 Set 对象。

```
scala> val set1=Set(1,2,2,4,5)
set1: scala.collection.immutable.Set[Int] = Set(1, 2, 4, 5)
```

C.3.5 映射

Scala 中的 Map 是个可变的键 / 值库，其他语言中也称为离散映射（hashmap）、字典（dictionary），与 Python 中的字典 dict 类似。Scala 并没有提供一个 {} 来定义 Map，创建 Map 时，指定键 - 值对为元组，也可以使用关系操作符（->）来指定键和值元组。与 Set、List 一样，Map 也是 Iterable 的一个子类型，支持与 List 相同操作。Map 默认是不可修改的，如果要成为可变 Map，需要显式定义其类型，如 scala.collection.mutable.Map。

```
scala> val m1=Map(1->"python",2->"java",4->"scala")
m1: scala.collection.immutable.Map[Int,String] = Map(1 -> python, 2 -> java, 4
-> scala)

scala> m1(4)
res49: String = scala

scala> m1+=(3->"spark")   // 因 m1 默认为不可变的，故添加元素报错
<console>:13: error: value += is not a member of scala.collection.immutable.
Map[Int,String]
       m1+=(3->"spark")
// 把 m1 修改为可变 Map
scala> val m1=scala.collection.mutable.Map(1->"python",2->"java",4->"scala")
m1: scala.collection.mutable.Map[Int,String] = Map(2 -> java, 4 -> scala, 1 ->
python)

scala> m1+=(3->"spark")
res51: m1.type = Map(2 -> java, 4 -> scala, 1 -> python, 3 -> spark)

scala> m1(5)="hadoop"

scala> m1
res53: scala.collection.mutable.Map[Int,String] = Map(2 -> java, 5 -> hadoop, 4
-> scala, 1 -> python, 3 -> spark)
```

Map 用于循环语句：

```
scala> for (key<-m1.keys) print(s"$key ")
2 5 4 1 3
scala> for (value<-m1.values) print(s"${value} ")
java hadoop scala python spark
```

C.3.6　集合的模式匹配

模式匹配是 Scala 的一个核心特性，这个特性可用于 Scala 的数据结构，利用该特性可大大简化代码、使逻辑显得更清晰。

```
scala> val rets=list1 match{
     | case x if x contains("spark")=>"big data tools"
     | case _ =>"no"
     | }
rets: String = big data tools
```

C.4　函数

函数是可重用的逻辑单位，在 Scala 中，函数是一等公民，函数式编程是 Scala 的一大特色。函数可以被赋值给一个变量，也可以作为一个函数的参数被传入，甚至还可以作为

函数的返回值返回。

C.4.1 函数定义

与 Python 一样，Scala 也使用 def 关键词来定义一个函数，定义函数一般需要确定函数的名称、参数和函数体，具体格式如下：

```
def <fun_name>(<param>:<type>[,...]):type=<expression>
```

如果有参数，需要说明参数类型，如果函数不是递归的，就不一定需要指明返回类型。Scala 编译时可以根据等号右侧的表达式的类型推导出返回类型。

如果函数体有多条语句，可用放在 {} 里，最后一句的值为函数返回值，有时需要用 return 显式指明返回值，并退出函数。

```
scala> def fun1(x:Int)=if(x>0) x else -x
fun1: (x: Int)Int
scala> fun1(-100) //调用函数
res0: Int = 100
//指明函数返回类型
scala> def fun2(x:Int):Int=if(x>0) x else -x
fun2: (x: Int)Int
scala> fun2(-10)
res1: Int = 10
```

当然参数不是必须的，有时函数没有参数，如下所示：

```
scala> def f1()="hello"
f1: ()String
scala> def f2="ok"
f2: String
scala> def f3:String="scala"
f3: String
```

C.4.2 匿名函数

Python 中 lambda 函数，Scala 有对应的匿名函数，匿名函数是 Scala 的重要特点之一，其语法格式为：

```
([<param1>:<type>,...])=><expression>
```

示例：

```
scala> (x:Int)=>x+2
res2: Int => Int = <function1>

scala> res2(10)
res3: Int = 12
//把函数赋值给变量 a
```

```
scala> var a=(x:Int)=>x+2
a: Int => Int = <function1>

scala> a(10)
res4: Int = 12

scala> val list1=List(2,5,7,8)
list1: List[Int] = List(2, 5, 7, 8)

scala> list1.filter((x:Int)=>x>6)
res5: List[Int] = List(7, 8)
// 省略参数类型
scala> list1.filter((x)=>x>6)
res6: List[Int] = List(7, 8)
// 单个参数，可去掉括号
scala> list1.filter(x=>x>6)
res7: List[Int] = List(7, 8)
// 如果一个参数，只使用一次，可用划线（_）代替参数，并只要表达式。
scala> list1.filter(_>6)
res8: List[Int] = List(7, 8)
```

C.4.3　递归函数

函数也是对象，可以把视为参数化表达式块，而表达式块是可以递归或嵌套，所以函数本身也可以递归或嵌套。以下通过实例说明如何实现函数的递归。

```
scala> // 计算 n !
scala> def fn(n:Int):Int={
     | if (n<=0)1
     | else n*fn(n-1)
     | }
fn: (n: Int)Int
scala> fn(3)
res1: Int = 6

scala> fn(5)
res2: Int = 120
```

但是递归会造成堆栈的大量占用，可以使用尾递归进行优化。优化的方式是，在函数定义前加上 @annotation.tailrec 这个函数注解（注不是注释）来声明尾递归，当编译器检测一个函数调用时尾递归时，它就覆盖当前的活动记录而不是在栈中去创建一个新的。只有最后一个语句为递归调用的函数才能由 scala 编译器完成尾递归优化，否则编译时将报错，还是以上例来说明，这是函数 fn 需要做些调整。

```
scala> @annotation.tailrec
     | def fn(n:Int,m:Int):Int={
     |       if (n<=0)m
     |       else fn(n-1,m*n)
```

```
     |       }
fn: (n: Int, m: Int)Int

scala> fn(5,1)
res3: Int = 120
```

C.4.4 有默认值参数的函数

Python 有带默认值参数的函数，Scala 中也有类似函数，使用了默认参数，你在调用函数的过程中可以输入该参数，这时函数就会调用它的默认参数值，如果传递了参数，则传递值会取代默认值。举例如下：

```
scala> def addInt(x:Int,y:Int=0):Int={
     | x+y
     | }
addInt: (x: Int, y: Int)Int
scala> addInt(20)    // 不输入有默认值的这个参数
res5: Int = 20

scala> addInt(20,30)  // 输入有默认值的这个参数
res6: Int = 50
```

C.4.5 变长参数的函数

在介绍 Python 函数时，我们介绍了变长参数的函数，函数定义格式为：

```
def 函数名 ( 参数 1, 参数 2,...,*tupl,**dict)
```

在 Scala 中也有类似功能的函数，而且更简洁，只需要在该函数参数类型后增加一个星号（*）即可，举例如下：

```
scala> def sum(items:Int*):Int={
    | var sum=0
    | for (i <- items) sum+=i
    | sum
    | }
sum: (items: Int*)Int

scala> sum(1,2,3)
res14: Int = 6

scala> sum(1,2,3,4,5)
res15: Int = 15
```

Python 可变参数可输入列表或字典等，以上方式将报错。此时，我们可以在调用函数参数后面追加 :_*,_* 操作符将告诉编译器把序列（Aarry,List,Seq,Vector 等）中的每个元素作为一个单独的参数传给函数，具体使用举例如下：

```
scala> val lit1=List(1,2,3,4,5)
lit1: List[Int] = List(1, 2, 3, 4, 5)

scala> sum(lit1:_*)
res21: Int = 15
```

C.4.6　部分应用的函数

调用函数时，通常需要指定函数中的所有参数（含默认值的参数除外），如果参数较多，而且调用时希望保留部分参数值，而修改部分参数值，该如何操作呢？

这时我们可以在原函数基础上定义一个部分应用函数即可，具体实现请看下例。

```
scala> def additems(x:Int,y:Int):Int=x+y
additems: (x: Int, y: Int)Int

scala> val additems10=additems(10,_:Int)
additems10: Int => Int = <function1>

scala> additems10(30)
res22: Int = 40
```

C.4.7　柯里化函数

实现部分应用函数，还有一种更简便的方法，采用多个参数表的函数，此时，不是将一个参数表分成应用参数和非应用参数，而是应用其中一个参数表的中参数，不用另一个参数表中参数，这种技术称为函数的柯里化（currying），具体实现如下：

```
scala> def additems(x:Int)(y:Int):Int=x+y
additems: (x: Int)(y: Int)Int

scala> val additems10=additems(10)_
additems10: Int => Int = <function1>

scala> additems10(30)
res23: Int = 40
```

C.4.8　遍历集合成员

在数据处理中经常一些函数遍历集合（如 Array、List、Tuple、Set、Map 等）中每个元素，函数遍历每个元素得到一个新的集合或没有返回值。经常使用的函数有 map、filter、foreach，reduce 等。这些函数在 Python 也有，称为内置函数。

```
scala> val list2=List.range(0,10,2)
list2: List[Int] = List(0, 2, 4, 6, 8)

scala> list2.map(i=>i+100)
```

```
res24: List[Int] = List(100, 102, 104, 106, 108)

scala> list2.foreach(i=>i+100)  //与 map 类似，但没有返回值
scala> list2.filter(i=>i>2)
res27: List[Int] = List(4, 6, 8)

scala> list2.reduce((x,y)=>x+y)
res28: Int = 20
```

上例中集合为一个列表，如何是 map，它又有哪些函数呢？

```
scala> val m1=Map(1->"scala",2->"python",3->"spark")
m1: scala.collection.immutable.Map[Int,String] = Map(1 -> scala, 2 -> python, 3
-> spark)
scala> val m2=m1.filterKeys(_>1)
m2: scala.collection.immutable.Map[Int,String] = Map(2 -> python, 3 -> spark)
scala> m1.foreach(x=>println(s"key is:${x._1},value is:${x._2}"))
key is:1,value is:scala
key is:2,value is:python
key is:3,value is:spark
```

作者简介

吴茂贵

资深BI和大数据专家，在BI、数据挖掘与分析、数据仓库、机器学习等领域有超过20年的工作经验，在Spark机器学习、TensorFlow深度学习领域有大量的实践经验。

郁明敏

对大数据、机器学习有一定的研究，擅长Python、Hadoop、Spark等技术，曾获得“江苏省TI杯大学生电子竞技大赛”二等奖和全国大学生数学建模大赛二等奖。

朱凤元

毕业于香港浸会大学，获运筹学与商业统计硕士学位，毕业后在飞牛网、永安保险、麦芽金服等公司从事数据挖掘建模工作。

张粤磊

资深DBA和大数据架构师，10余年一线数据挖掘与分析实战经验。先后在咨询、金融、互联网行业担任数据平台技术负责人或架构师。

杨本法

高级算法工程师，在机器学习、文本挖掘、可视化等领域有多年实践经验。熟悉Hadoop生态圈的相关技术，在R、Spark方面有丰富的实战经验。

封面设计：王建敏

Inside Explore the Spark Machine Learning

深度实践 Spark机器学习



本书通过系统介绍Spark平台及其上的机器学习工具，能帮助读者快速了解机器学习的基础知识，以及如何利用Spark平台和Spark ML工具快速搭建机器学习模块，在实战中提升数据分析和建模的能力。此外，针对时下最火爆的深度学习，本书重点讲解了TensorFlowOnSpark，其整合了深度学习框架TensorFlow和Spark，读者可以在实践中了解深度学习的基础知识及实操效果。

—— 肖京　国家千人计划专家，平安科技首席科学家兼智能引擎部总经理，美国卡内基梅隆大学计算机博士

本书能让读者深入了解机器学习的原理，通过 Spark 框架实践机器学习的整个流程，包括算法的封装、特征转换、特征选择、数据清洗、数据划分、模型评估等。此外，还重点讲解了机器学习过程的流程化、标准化、规范化。本书采用循序渐进的方式介绍了 Spark ML，重点介绍了TensorFlowOnSpark，包括深度学习框架TensorFlow的基础知识，以及使用卷积神经网络、循环神经网络等的深度学习案例，是一本非常难得的机器学习实践指南。

—— 卢亿雷　大数据资深专家，精硕科技（AdMaster）技术副总裁兼总架构师，CCF（中国计算学会）大数据专委委员，北京航空航天大学特聘教授

本书详细讲解了Spark框架下的机器学习和深度学习的技术与实践方法，特别注重二者在实际应用环境下的实践，实用性极强。

—— 谭昶　科大讯飞股份有限公司大数据研究院副院长，中国计算机学会大数据专家委员会委员

本书以机器学习系统架构和Spark框架为主线，渐进式地讲解了机器学习概念、技术和关键步骤。理论与实践紧密结合，既全面，又精练。在最后部分，与时俱进地延伸出深度学习的部分，以TensorFlow的实例，讲解了卷积神经网络和循环神经网络，并以TensorFlowOnSpark为结尾来弥补Spark在深度学习方面的不足。本书能引导读者以实践来领悟和夯实机器学习的专业知识和实践技能，完成数据时代的“软件”迭代升级，是一本以实战为抓手的好书。

—— 王冬　博世中国IT数据经理

上架指导：计算机/人工智能
ISBN 978-7-111-58995-2
9 787111 589952
定价：69.00元

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn